The **java.util** package contains one of Java's most powerful subsystems: collections. Collections were added by the initial release of Java 2, and enhanced by Java 2, version 1.4. A *collection* is a group of objects. The addition of collections caused fundamental alterations in the structure and architecture of many elements in **java.util**. It also expanded the domain of tasks to which the package can be applied. Collections are a state-of-the-art technology that merits close attention by all Java programmers.

In addition to collections, **java.util** contains a wide assortment of classes and interfaces that support a broad range of functionality. These classes and interfaces are used throughout the core Java packages and, of course, are also available for use in programs that you write. Their applications include generating pseudorandom numbers, manipulating date and time, observing events, manipulating sets of bits, and tokenizing strings. Because of its many features, **java.util** is one of Java's most widely used packages.

The **java.util** classes are listed here.

AbstractCollection (Java 2)	EventObject	PropertyResourceBundle
AbstractList (Java 2)	GregorianCalendar	Random
AbstractMap (Java 2)	HashMap (Java 2)	ResourceBundle
AbstractSequentialList (Java 2)	HashSet (Java 2)	SimpleTimeZone
AbstractSet (Java 2)	Hashtable	Stack
ArrayList (Java 2)	IdentityHashMap (Java 2, v1.4)	StringTokenizer
Arrays (Java 2)	LinkedHashMap (Java 2, v1.4)	Timer (Java 2, v1.3)
BitSet	LinkedHashSet (Java 2, v1.4)	TimerTask (Java 2, v1.3)
Calendar	LinkedList (Java 2)	TimeZone
Collections (Java 2)	ListResourceBundle	TreeMap (Java 2)
Currency (Java 2, v1.4)	Locale	TreeSet (Java 2)
Date	Observable	Vector
Dictionary	Properties	WeakHashMap (Java 2)
EventListenerProxy (Java 2, v1.4)	PropertyPermission (Java 2)	

java.util defines the following interfaces. Notice that most were added by Java 2.

Collection (Java 2)	List (Java 2)	RandomAccess (Java 2, v1.4)
Comparator (Java 2)	ListIterator (Java 2)	Set (Java 2)
Enumeration	Map (Java 2)	SortedMap (Java 2)
EventListener	Map.Entry (Java 2)	SortedSet (Java 2)
Iterator (Java 2)	Observer	

The **ResourceBundle**, **ListResourceBundle**, and **PropertyResourceBundle** classes aid in the internationalization of large programs with many locale-specific resources. These classes are not examined here. **PropertyPermission**, which allows you to grant a read/write permission to a system property, is also beyond the scope of this book. **EventObject**, **EventListener**, and **EventListenerProxy** are described in Chapter 20. The remaining classes and interfaces are examined in detail.

Because **java.util** is quite large, its description is broken into two chapters. This chapter examines those members of **java.util** that relate to collections of objects. Chapter 16 discusses the other classes and interfaces.

Collections Overview

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 manipulate groups of objects. Although these classes were quite useful, they lacked a central, unifying theme. Thus, the way that you used **Vector** was different from the way that you used **Properties**, for example. Also, the previous, ad hoc approach was not designed to be easily extensible or adaptable. Collections are an answer to these (and other) problems.

The collections framework was designed to meet several goals. First, the framework had to be high-performance. The implementations for the fundamental collections (dynamic arrays, linked lists, trees, and hash tables) are highly efficient. You seldom, if ever, need to code one of these "data engines" manually. Second, the framework had to allow different types of collections to work in a similar manner and with a high degree of interoperability. Third, extending and/or adapting a collection had to be easy. Toward 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 provided that you may use as-is. You may also implement your own collection, if you choose. Various special-purpose implementations are created for your convenience, and some partial implementations are provided that make creating your own collection class easier. Finally, mechanisms were added that allow the integration of standard arrays into the collections framework.

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. Each collection class need not implement its own versions. 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**. 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, the framework defines several map interfaces and classes. *Maps* store key/value pairs. Although maps are not "collections" in the proper use of the term, they are fully 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, if you choose.

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. It is important to understand that 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 of doing several things.

One last thing: If you are familiar with C++, then you will find it helpful 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 a collection.

The Collection Interfaces

The collections framework defines several interfaces. This section provides an overview of each interface. Beginning with the collection interfaces is necessary because they determine the fundamental nature of the collection classes. Put differently, the concrete classes simply provide different implementations of the standard interfaces. The interfaces that underpin collections are summarized in the following table:

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

In addition to the collection interfaces, collections also use the **Comparator**, **Iterator**, **ListIterator** and **RandomAccess** interfaces, which are described in depth later in this chapter. Briefly, **Comparator** defines how two objects are compared; **Iterator** and **ListIterator** enumerate the objects within a collection. By implementing **RandomAccess**, a list indicates that it supports efficient, random access to its elements.

To provide the greatest flexibility in their use, the collection interfaces allow some methods to be optional. The optional methods enable you to modify the contents of a collection. Collections that support these methods are called *modifiable*. Collections that do not allow their contents to be changed are called *unmodifiable*. If an attempt is made

to use one of these methods on an unmodifiable collection, an **UnsupportedOperationException** is thrown. All the built-in collections are modifiable.

The following sections examine the collection interfaces.

The Collection Interface

The **Collection** interface is the foundation upon which the collections framework is built. It declares the core methods that all collections will have. These methods are summarized in Table 15-1. Because all collections implement **Collection**, familiarity with its methods is necessary for a clear understanding of the framework. Several of these methods can throw an **UnsupportedOperationException**. As explained, this occurs if a collection cannot be modified. 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.

Method	Description
boolean add(Object *obj*)	Adds *obj* to the invoking collection. Returns **true** if *obj* was added to the collection. Returns **false** if *obj* is already a member of the collection, or if the collection does not allow duplicates.
boolean addAll(Collection *c*)	Adds all the elements of *c* to the invoking collection. Returns **true** if the operation succeeded (i.e., the elements were added). Otherwise, returns **false**.
void clear( )	Removes all elements from the invoking collection.
boolean contains(Object *obj*)	Returns **true** if *obj* is an element of the invoking collection. Otherwise, returns **false**.
boolean containsAll(Collection *c*)	Returns **true** if the invoking collection contains all elements of *c*. Otherwise, returns **false**.
boolean equals(Object *obj*)	Returns **true** if the invoking collection and *obj* are equal. Otherwise, returns **false**.
int hashCode( )	Returns the hash code for the invoking collection.

Table 15-1. *The Methods Defined by* Collection

Method	Description
boolean isEmpty( )	Returns **true** if the invoking collection is empty. Otherwise, returns **false**.
Iterator iterator( )	Returns an iterator for the invoking collection.
boolean remove(Object *obj*)	Removes one instance of *obj* from the invoking collection. Returns **true** if the element was removed. Otherwise, returns **false**.
boolean removeAll(Collection *c*)	Removes all elements of *c* from the invoking collection. Returns **true** if the collection changed (i.e., elements were removed). Otherwise, returns **false**.
boolean retainAll(Collection *c*)	Removes all elements from the invoking collection except those in *c*. Returns **true** if the collection changed (i.e., elements were removed). Otherwise, returns **false**.
int size( )	Returns the number of elements held in the invoking collection.
Object[ ] toArray( )	Returns an array that contains all the elements stored in the invoking collection. The array elements are copies of the collection elements.
Object[ ] toArray(Object *array*[ ])	Returns an array containing only those collection elements whose type matches that of *array*. The array elements are copies of the collection elements. If the size of *array* equals the number of matching elements, these are returned in *array*. If the size of *array* is less than the number of matching elements, a new array of the necessary size is allocated and returned. If the size of *array* is greater than the number of matching elements, the array element following the last collection element is set to **null**. An **ArrayStoreException** is thrown if any collection element has a type that is not a subtype of *array*.

Table 15-1. *The Methods Defined by* Collection (continued)

Objects are added to a collection by calling **add( )**. Notice that **add( )** takes an argument of type **Object**. Because **Object** is a superclass of all classes, any type of object may be stored in a collection. However, primitive types may not. For example, a collection cannot directly store values of type **int**, **char**, **double**, and so forth. Of course, if you want to store such objects, you can also use one of the primitive type wrappers described in Chapter 14. You can add the entire contents of one collection to another by calling **addAll( )**.

You can remove an object by using **remove( )**. To remove a group of objects, call **removeAll( )**. You can remove all elements except those of a specified group by calling **retainAll( )**. To empty a collection, call **clear( )**.

You can determine whether a collection contains a specific object by calling **contains( )**. To determine whether one collection contains all the members of another, call **containsAll( )**. You can determine when a collection is empty by calling **isEmpty( )**. The number of elements currently held in a collection can be determined by calling **size( )**.

The **toArray( )** method returns an array that contains the elements stored in the invoking collection. This method is more important than it might at first seem. Often, processing the contents of a collection by using array-like syntax is advantageous. By providing a pathway between collections and arrays, you can have the best of both worlds.

Two collections can be compared for equality by calling **equals( )**. The precise meaning of "equality" may differ from collection to collection. For example, you can implement **equals( )** so that it compares the values of elements stored in the collection. Alternatively, **equals( )** can compare references to those elements.

One more very important method is **iterator( )**, which returns an iterator to a collection. As you will see, iterators are crucial to successful programming when using the collections framework.

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, which are summarized in Table 15-2. Note again that 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.

To the versions of **add( )** and **addAll( )** defined by **Collection**, **List** adds the methods **add(int, Object)** and **addAll(int, Collection)**. These methods insert elements at the specified index. Also, the semantics of **add(Object)** and **addAll(Collection)** defined by **Collection** are changed by **List** so that they add elements to the end of the list.

To obtain the object stored at a specific location, call **get( )** with the index of the object. To assign a value to an element in the list, call **set( )**, specifying the index of the object to be changed. To find the index of an object, use **indexOf( )** or **lastIndexOf( )**.

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 from *start* to *end*–1 in the invoking list. Elements in the returned list are also referenced by the invoking object.

Table 15-2. *The Methods Defined by List*

You can obtain a sublist of a list by calling **subList( )**, specifying the beginning and ending indexes of the sublist. As you can imagine, **subList( )** makes list processing quite convenient.

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 the methods summarized in Table 15-3. 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( )**.

Method	Description
Comparator comparator( )	Returns the invoking sorted set's comparator. If the natural ordering is used for this set, **null** is returned.
Object first( )	Returns the first element in the invoking sorted set.
SortedSet headSet(Object *end*)	Returns a **SortedSet** containing those elements less than *end* that are contained in the invoking sorted set. Elements in the returned sorted set are also referenced by the invoking sorted set.
Object last( )	Returns the last element in the invoking sorted set.

Table 15-3. *The Methods Defined by* SortedSet

Method	Description
SortedSet subSet(Object *start*, Object *end*)	Returns a **SortedSet** that includes those elements between *start* and *end*–1. Elements in the returned collection are also referenced by the invoking object.
SortedSet tailSet(Object *start*)	Returns a **SortedSet** that contains those elements greater than or equal to *start* that are contained in the sorted set. Elements in the returned set are also referenced by the invoking object.

Table 15-3. *The Methods Defined by* SortedSet (continued)

The Collection Classes

Now that you are familiar with the collection interfaces, you are ready to examine the standard classes that implement them. Some of the classes provide full implementations that can be used as-is. Others are abstract, providing skeletal implementations that are used as starting points for creating concrete collections. None of the collection classes are synchronized, but as you will see later in this chapter, it is possible to obtain synchronized versions.

The standard collection classes are summarized in the following table:

Class	Description
AbstractCollection	Implements most of the **Collection** interface.
AbstractList	Extends **AbstractCollection** and implements most of the **List** interface.
AbstractSequentialList	Extends **AbstractList** for use by a collection that uses sequential rather than random access of its elements.
LinkedList	Implements a linked list by extending **AbstractSequentialList**.
ArrayList	Implements a dynamic array by extending **AbstractList**.

Class	Description
AbstractSet	Extends **AbstractCollection** and implements most of the **Set** interface.
HashSet	Extends **AbstractSet** for use with a hash table.
LinkedHashSet	Extends **HashSet** to allow insertion-order iterations.
TreeSet	Implements a set stored in a tree. Extends **AbstractSet**.

Note *In addition to the collection classes, several legacy classes, such as* **Vector**, **Stack**, *and* **Hashtable**, *have been reengineered to support collections. These are examined later in this chapter.*

The following sections examine the concrete collection classes and illustrate their use.

The ArrayList Class

The **ArrayList** class extends **AbstractList** and implements the **List** interface. **ArrayList** supports dynamic arrays that can grow as needed. In Java, standard arrays are of a fixed length. After arrays are created, they cannot grow or shrink, which means that you must know in advance how many elements an array will hold. But, sometimes, you may not know until run time precisely how large of an array you need. To handle this situation, the collections framework defines **ArrayList**. In essence, an **ArrayList** is a variable-length array of object references. That is, an **ArrayList** can dynamically increase or decrease in size. Array lists are created with an initial size. When this size is exceeded, the collection is automatically enlarged. When objects are removed, the array may be shrunk.

Note *Dynamic arrays are also supported by the legacy class* **Vector**, *which is described later in this chapter.*

ArrayList has the constructors shown here:

```
ArrayList( )
ArrayList(Collection c)
ArrayList(int capacity)
```

The first constructor builds an empty array list. The second constructor builds an array list that is initialized with the elements of the collection *c*. The third constructor builds an array list that has the specified initial *capacity*. The capacity is the size of the underlying array that is used to store the elements. The capacity grows automatically as elements are added to an array list.

The following program shows a simple use of **ArrayList**. An array list is created, and then objects of type **String** are added to it. (Recall that a quoted string is translated into a **String** object.) The list is then displayed. Some of the elements are removed and the list is displayed again.

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

class ArrayListDemo {
  public static void main(String args[]) {
    // create an array list
    ArrayList al = new ArrayList();

    System.out.println("Initial size of al: " +
                       al.size());

    // add elements to the array list
    al.add("C");
    al.add("A");
    al.add("E");
    al.add("B");
    al.add("D");
    al.add("F");
    al.add(1, "A2");

    System.out.println("Size of al after additions: " +
                       al.size());

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

    // Remove elements from the array list
    al.remove("F");
    al.remove(2);

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

The output from this program is shown here:

```
Initial size of al: 0
Size of al after additions: 7
```

```
Contents of al: [C, A2, A, E, B, D, F]
Size of al after deletions: 5
Contents of al: [C, A2, E, B, D]
```

Notice that **al** starts out empty and grows as elements are added to it. When elements are removed, its size is reduced.

In the preceding example, the contents of a collection are displayed using the default conversion provided by **toString( )**, which was inherited from **AbstractCollection**. Although it is sufficient for short, sample programs, you seldom use this method to display the contents of a real-world collection. Usually, you provide your own output routines. But, for the next few examples, the default output created by **toString( )** will continue to be used.

Although the capacity of an **ArrayList** object increases automatically as objects are stored in it, you can increase the capacity of an **ArrayList** object manually by calling **ensureCapacity( )**. You might want to do this if you know in advance that you will be storing many more items in the collection that it can currently hold. By increasing its capacity once, at the start, you can prevent several reallocations later. Because reallocations are costly in terms of time, preventing unnecessary ones improves performance. The signature for **ensureCapacity( )** is shown here:

```
void ensureCapacity(int cap)
```

Here, *cap* is the new capacity.

Conversely, if you want to reduce the size of the array that underlies an **ArrayList** object so that it is precisely as large as the number of items that it is currently holding, call **trimToSize( )**, shown here:

```
void trimToSize( )
```

Obtaining an Array from an ArrayList

When working with **ArrayList**, you will sometimes want to obtain an actual array that contains the contents of the list. As explained earlier, you can do this by calling **toArray( )**. Several reasons exist why you might want to convert a collection into an array such as:

- To obtain faster processing times for certain operations.
- To pass an array to a method that is not overloaded to accept a collection.
- To integrate your newer, collection-based code with legacy code that does not understand collections.

Whatever the reason, converting an **ArrayList** to an array is a trivial matter, as the following program shows:

```
// Convert an ArrayList into an array.
import java.util.*;
```

```
class ArrayListToArray {
  public static void main(String args[]) {
    // Create an array list
    ArrayList al = new ArrayList();

    // Add elements to the array list
    al.add(new Integer(1));
    al.add(new Integer(2));
    al.add(new Integer(3));
    al.add(new Integer(4));

    System.out.println("Contents of al: " + al);

    // get array
    Object ia[] = al.toArray();
    int sum = 0;

    // sum the array
    for(int i=0; i<ia.length; i++)
      sum += ((Integer) ia[i]).intValue();

    System.out.println("Sum is: " + sum);
  }
}
```

The output from the program is shown here:

```
Contents of al: [1, 2, 3, 4]
Sum is: 10
```

The program begins by creating a collection of integers. As explained, you cannot store primitive types in a collection, so objects of type **Integer** are created and stored. Next, **toArray( )** is called and it obtains an array of **Objects**. The contents of this array are cast to **Integer**, and then the values are summed.

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:

LinkedList( )
LinkedList(Collection *c*)

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( )**. Their signatures are shown here:

```
void addFirst(Object obj)
void addLast(Object obj)
```

Here, *obj* is the item being added.

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

```
Object getFirst( )
Object getLast( )
```

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

```
Object removeFirst( )
Object removeLast( )
```

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

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

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

    // add elements to the linked list
    ll.add("F");
    ll.add("B");
    ll.add("D");
    ll.add("E");
    ll.add("C");
    ll.addLast("Z");
    ll.addFirst("A");

    ll.add(1, "A2");
```

```
        System.out.println("Original contents of ll: " + ll);

        // remove elements from the linked list
        ll.remove("F");
        ll.remove(2);

        System.out.println("Contents of ll after deletion: "
                           + ll);

        // remove first and last elements
        ll.removeFirst();
        ll.removeLast();

        System.out.println("ll after deleting first and last: "
                           + ll);

        // get and set a value
        Object val = ll.get(2);
        ll.set(2, (String) val + " Changed");

        System.out.println("ll after change: " + ll);
    }
}
```

The output from this program is shown here:

```
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]
```

Because **LinkedList** implements the **List** interface, calls to **add(Object)** append items to the end of the list, as does **addLast( )**. To insert items at a specific location, use the **add(int, Object)** form of **add( )**, as illustrated by the call to **add(1, "A2")** in the example.

Notice how the third element in ll is changed by employing calls to **get( )** and **set( )**. To obtain the current value of an element, pass **get( )** the index at which the element is stored. To assign a new value to that index, pass **set( )** the index and its new value.

The HashSet Class

HashSet extends **AbstractSet** and implements the **Set** interface. It creates a collection that uses a hash table for storage. As most readers likely know, a hash table stores

information by using a mechanism called hashing. In *hashing*, the informational content of a key is used to determine a unique value, called its *hash code*. The hash code is then used as the index at which the data associated with the key is stored. The transformation of the key into its hash code is performed automatically—you never see the hash code itself. Also, your code can't directly index the hash table. The advantage of hashing is that it allows the execution time of basic operations, such as **add( )**, **contains( )**, **remove( )**, and **size( )**, to remain constant even for large sets.

The following constructors are defined:

HashSet( )
HashSet(Collection *c*)
HashSet(int *capacity*)
HashSet(int *capacity*, float *fillRatio*)

The first form constructs a default hash set. The second form initializes the hash set by using the elements of *c*. The third form initializes the capacity of the hash set to *capacity*. The fourth form initializes both the capacity and the fill ratio (also called *load capacity*) of the hash set from its arguments. The fill ratio must be between 0.0 and 1.0, and it determines how full the hash set can be before it is resized upward. Specifically, when the number of elements is greater than the capacity of the hash set multiplied by its fill ratio, the hash set is expanded. For constructors that do not take a fill ratio, 0.75 is used.

HashSet does not define any additional methods beyond those provided by its superclasses and interfaces.

Importantly, note that a hash set does not guarantee the order of its elements, because the process of hashing doesn't usually lend itself to the creation of sorted sets. If you need sorted storage, then another collection, such as **TreeSet**, is a better choice.

Here is an example that demonstrates **HashSet**:

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

class HashSetDemo {
  public static void main(String args[]) {
    // create a hash set
    HashSet hs = new HashSet();

    // add elements to the hash set
    hs.add("B");
    hs.add("A");
    hs.add("D");
    hs.add("E");
    hs.add("C");
    hs.add("F");
```

Method	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( )**.

Table 15-4. *The Methods Declared by* Iterator

Method	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( )**.

Table 15-5. *The Methods Declared by* ListIterator

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**. As explained, a list iterator 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+
```

Pay special attention to how the list is displayed in reverse. After the list is modified, **litr** points to the end of the list. (Remember, **litr.hasNext( )** returns false when the end of the list has been reached.) To traverse the list in reverse, the program continues to use **litr**, but this time it checks to see whether it has a previous element. As long as it does, that element is obtained and displayed.

Storing User-Defined Classes in Collections

For the sake of simplicity, the foregoing examples have stored built-in objects, such as **String** or **Integer**, in a collection. Of course, collections are not limited to the storage of built-in objects. Quite the contrary. The power of collections is that they can store any type of object, including objects of classes that you create. For example, consider the following example that uses a **LinkedList** to store mailing addresses:

```
// A simple mailing list example.
import java.util.*;
```

```
class Address {
  private String name;
  private String street;
  private String city;
  private String state;
  private String code;

  Address(String n, String s, String c,
          String st, String cd) {
    name = n;
    street = s;
    city = c;
    state = st;
    code = cd;
  }

  public String toString() {
    return name + "\n" + street + "\n" +
           city + " " + state + " " + code;
  }
}

class MailList {
  public static void main(String args[]) {
    LinkedList ml = new LinkedList();

    // add elements to the linked list
    ml.add(new Address("J.W. West", "11 Oak Ave",
                       "Urbana", "IL", "61801"));
    ml.add(new Address("Ralph Baker", "1142 Maple Lane",
                       "Mahomet", "IL", "61853"));
    ml.add(new Address("Tom Carlton", "867 Elm St",
                       "Champaign", "IL", "61820"));

    Iterator itr = ml.iterator();
    while(itr.hasNext()) {
      Object element = itr.next();
      System.out.println(element + "\n");
    }
    System.out.println();
  }
}
```

The output from the program is shown here:

```
J.W. West
11 Oak Ave
Urbana IL 61801

Ralph Baker
1142 Maple Lane
Mahomet IL 61853

Tom Carlton
867 Elm St
Champaign IL 61820
```

Aside from storing a user-defined class in a collection, another important thing to notice about the preceding program is that it is quite short. When you consider that it sets up a linked list that can store, retrieve, and process mailing addresses in about 50 lines of code, the power of the collections framework begins to become apparent. As most readers know, if all of this functionality had to be coded manually, the program would be several times longer. Collections offer off-the-shelf solutions to a wide variety of programming problems. You should use them whenever the situation presents itself.

The RandomAccess Interface

Java 2, version 1.4 adds the **RandomAccess** interface. This interface contains no members. However, by implementing this interface, a collection signals that it supports efficient random access to its elements. Although a collection might support random access, it might not do so efficiently. By checking for the **RandomAccess** interface, client code can determine at run time whether a collection is suitable for certain types of random access operations—especially as they apply to large collections. (You can use **instanceof** to determine if a class implements an interface.) **RandomAccess** is implemented by **ArrayList** and by the legacy **Vector** class.

Working with Maps

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

Because the map interfaces define the character and nature of maps, this discussion of maps begins with them. The following interfaces support maps:

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.

Each interface is examined next, in turn.

The Map Interface

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. The methods declared by **Map** are summarized in Table 15-6. 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**.

Table 15-6. *The Methods Defined by* Map

Method	Description
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 the invoking 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.

Table 15-6. *The Methods Defined by* Map (continued)

Maps revolve around 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.

As mentioned earlier, 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. The methods declared by **SortedMap** are summarized in Table 15-7. 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 are less 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*.

Table 15-7. *The Methods Defined by* SortedMap

The Map.Entry Interface

The **Map.Entry** interface enables you to work with a map entry. Recall that the **entrySet( )** method declared by the **Map** interface returns a **Set** containing the map entries. Each of these set elements is a **Map.Entry** object. Table 15-8 summarizes the methods declared by this interface.

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.

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.

Table 15-8. *The Methods Defined by* Map.Entry

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. This class is not discussed further here. The others are described next.

The HashMap Class

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

The following constructors are defined:

HashMap( )
HashMap(Map *m*)
HashMap(int *capacity*)
HashMap(int *capacity*, float *fillRatio*)

The first form constructs a default hash map. The second form initializes the hash map by using the elements of *m*. The third form initializes the capacity of the hash map to *capacity*. The fourth form initializes both the capacity and fill ratio of the hash map by using its arguments. The meaning of capacity and fill ratio is the same as for **HashSet**, described earlier.

HashMap implements **Map** and extends **AbstractMap**. It does not add any methods of its own.

You should note that a hash map does *not* guarantee the order of its elements. Therefore, the order in which elements are added to a hash map is not necessarily the order in which they are read by an iterator.

The following program illustrates **HashMap**. It maps names to account balances. Notice how a set-view is obtained and used.

```
import java.util.*;

class HashMapDemo {
  public static void main(String args[]) {

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

    // Put elements to the map
    hm.put("John Doe", new Double(3434.34));
    hm.put("Tom Smith", new Double(123.22));
    hm.put("Jane Baker", new Double(1378.00));
    hm.put("Todd Hall", new Double(99.22));
    hm.put("Ralph Smith", new Double(-19.08));
```

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

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

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

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

Output from this program is shown here (the precise order may vary).

```
Todd Hall: 99.22
Ralph Smith: -19.08
John Doe: 3434.34
Jane Baker: 1378.0
Tom Smith: 123.22

John Doe's current balance: 4434.34
```

The program begins by creating a hash map and then adds the mapping of names to balances. Next, the contents of the map are displayed by using a set-view, obtained by calling **entrySet( )**. The keys and values are displayed by calling the **getKey( )** and **getValue( )** methods that are defined by **Map.Entry**. Pay close attention to how the deposit is made into John Doe's account. The **put( )** method automatically replaces any preexisting value that is associated with the specified key with the new value. Thus, after John Doe's account is updated, the hash map will still contain just one "John Doe" account.

The TreeMap Class

The **TreeMap** class implements the **Map** interface by using a tree. A **TreeMap** provides an efficient means of storing key/value pairs in sorted order, and allows rapid

retrieval. You should note that, unlike a hash map, a tree map guarantees that its elements will be sorted in ascending key order.

The following **TreeMap** constructors are defined:

TreeMap( )
TreeMap(Comparator *comp*)
TreeMap(Map *m*)
TreeMap(SortedMap *sm*)

The first form constructs an empty tree map that will be sorted by using the natural order of its keys. The second form constructs an empty tree-based map that will be sorted by using the **Comparator** *comp*. (Comparators are discussed later in this chapter.) The third form initializes a tree map with the entries from *m*, which will be sorted by using the natural order of the keys. The fourth form initializes a tree map with the entries from *sm*, which will be sorted in the same order as *sm*.

TreeMap implements **SortedMap** and extends **AbstractMap**. It does not define any additional methods of its own.

The following program reworks the preceding example so that it uses **TreeMap**:

```
import java.util.*;

class TreeMapDemo {
  public static void main(String args[]) {

    // Create a tree map
    TreeMap tm = new TreeMap();

    // Put elements to the map
    tm.put("John Doe", new Double(3434.34));
    tm.put("Tom Smith", new Double(123.22));
    tm.put("Jane Baker", new Double(1378.00));
    tm.put("Todd Hall", new Double(99.22));
    tm.put("Ralph Smith", new Double(-19.08));

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

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

    // Display elements
    while(i.hasNext()) {
      Map.Entry me = (Map.Entry)i.next();
```

```
      System.out.print(me.getKey() + ": ");
      System.out.println(me.getValue());
    }
    System.out.println();

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

The following is the output from this program:

```
Jane Baker: 1378.0
John Doe: 3434.34
Ralph Smith: -19.08
Todd Hall: 99.22
Tom Smith: 123.22

John Doe's current balance: 4434.34
```

Notice that **TreeMap** sorts the keys. However, in this case, they are sorted by first name instead of last name. You can alter this behavior by specifying a comparator when the map is created, as described shortly.

The LinkedHashMap Class

Java 2, version 1.4 adds the **LinkedHashMap** class. This class extends **HashMap**. **LinkedHashMap** maintains a linked list of the entries in the map, in the order in which they were inserted. This allows insertion-order iteration over the map. That is, when iterating a **LinkedHashMap**, the elements will be returned in the order in which they were inserted. You can also create a **LinkedHashMap** that returns its elements in the order in which they were last accessed.

LinkedHashMap defines the following constructors.

LinkedHashMap( )
LinkedHashMap(Map *m*)
LinkedHashMap(int *capacity*)
LinkedHashMap(int *capacity*, float *fillRatio*)
LinkedHashMap(int *capacity*, float *fillRatio*, boolean *Order*)

The first form constructs a default **LinkedHashMap**. The second form initializes the **LinkedHashMap** with the elements from *m*. The third form initializes the capacity. The fourth form initializes both capacity and fill ratio. The meaning of capacity and fill ratio are the same as for **HashMap**. The last form allows you to specify whether the elements will be stored in the linked list by insertion order, or by order of last access. If *Order* is **true**, then access order is used. If *Order* is **false**, then insertion order is used.

LinkedHashMap adds only one method to those defined by **HashMap**. This method is **removeEldestEntry( )** and it is shown here.

```
protected boolean removeEldestEntry(Map.Entry e)
```

This method is called by **put( )** and **putAll( )**. The oldest entry is passed in *e*. By default, this method returns **false** and does nothing. However, if you override this method, then you can have the **LinkedHashMap** remove the oldest entry in the map. To do this, have your override return **true**. To keep the oldest entry, return **false**.

The IdentityHashMap Class

Java 2, version 1.4 adds the **IdentityHashMap** class. This class implements **AbstractMap**. It is similar to **HashMap** except that it uses reference equality when comparing elements. The Java 2 documentation explicitly states that **IdentityHashMap** is not for general use.

Arrays

The **Arrays** class provides various methods that are useful when working with arrays. Although these methods technically aren't part of the collections framework, they help bridge the gap between collections and arrays. **Arrays** was added by Java 2. Each method defined by **Arrays** is examined in this section.

The **asList( )** method returns a **List** that is backed by a specified array. In other words, both the list and the array refer to the same location. It has the following signature:

```
static List asList(Object[ ] array)
```

Here, *array* is the array that contains the data.

The **binarySearch( )** method uses a binary search to find a specified value. This method must be applied to sorted arrays. It has the following forms:

static int binarySearch(byte[ ] *array*, byte *value*)
static int binarySearch(char[ ] *array*, char *value*)
static int binarySearch(double[ ] *array*, double *value*)
static int binarySearch(float[ ] *array*, float *value*)
static int binarySearch(int[ ] *array*, int *value*)
static int binarySearch(long[ ] *array*, long *value*)
static int binarySearch(short[ ] *array*, short *value*)
static int binarySearch(Object[ ] *array*, Object *value*)
static int binarySearch(Object[ ] *array*, Object *value*, Comparator *c*)

Here, *array* is the array to be searched and *value* is the value to be located. The last two forms throw a **ClassCastException** if *array* contains elements that cannot be compared (for example, **Double** and **StringBuffer**) or if *value* is not compatible with the types in *array*. In the last form, the **Comparator** *c* is used to determine the order of the elements in *array*. In all cases, if *value* exists in *array*, the index of the element is returned. Otherwise, a negative value is returned.

The **equals( )** method returns **true** if two arrays are equivalent. Otherwise, it returns **false**. The **equals( )** method has the following forms:

static boolean equals(boolean *array1*[ ], boolean *array2*[ ])
static boolean equals(byte *array1*[ ], byte *array2*[ ])
static boolean equals(char *array1*[ ], char *array2*[ ])
static boolean equals(double *array1*[ ], double *array2*[ ])
static boolean equals(float *array1*[ ], float *array2*[ ])
static boolean equals(int *array1*[ ], int *array2*[ ])
static boolean equals(long *array1*[ ], long *array2*[ ])
static boolean equals(short *array1*[ ], short *array2*[ ])
static boolean equals(Object *array1*[ ], Object *array2*[ ])

Here, *array1* and *array2* are the two arrays that are compared for equality.

The **fill( )** method assigns a value to all elements in an array. In other words, it fills an array with a specified value. The **fill( )** method has two versions. The first version, which has the following forms, fills an entire array:

static void fill(boolean *array*[ ], boolean *value*)
static void fill(byte *array*[ ], byte *value*)
static void fill(char *array*[ ], char *value*)
static void fill(double *array*[ ], double *value*)
static void fill(float *array*[ ], float *value*)
static void fill(int *array*[ ], int *value*)
static void fill(long *array*[ ], long *value*)
static void fill(short *array*[ ], short *value*)
static void fill(Object *array*[ ], Object *value*)

Here, *value* is assigned to all elements in *array*.

The second version of the **fill( )** method assigns a value to a subset of an array. Its forms are shown here:

```
static void fill(boolean array[ ], int start, int end, boolean value)
static void fill(byte array[ ], int start, int end, byte value)
static void fill(char array[ ], int start, int end, char value)
static void fill(double array[ ], int start, int end, double value)
static void fill(float array[ ], int start, int end, float value)
static void fill(int array[ ], int start, int end, int value)
static void fill(long array[ ], int start, int end, long value)
static void fill(short array[ ], int start, int end, short value)
static void fill(Object array[ ], int start, int end, Object value)
```

Here, *value* is assigned to the elements in *array* from position *start* to position *end*–1. These methods may all throw an **IllegalArgumentException** if *start* is greater than *end*, or an **ArrayIndexOutOfBoundsException** if *start* or *end* is out of bounds.

The **sort( )** method sorts an array so that it is arranged in ascending order. The **sort( )** method has two versions. The first version, shown here, sorts the entire array:

```
static void sort(byte array[ ])
static void sort(char array[ ])
static void sort(double array[ ])
static void sort(float array[ ])
static void sort(int array[ ])
static void sort(long array[ ])
static void sort(short array[ ])
static void sort(Object array[ ])
static void sort(Object array[ ], Comparator c)
```

Here, *array* is the array to be sorted. In the last form, *c* is a **Comparator** that is used to order the elements of *array*. The forms that sort arrays of **Object** can also throw a **ClassCastException** if elements of the array being sorted are not comparable.

The second version of **sort( )** enables you to specify a range within an array that you want to sort. Its forms are shown here:

```
static void sort(byte array[ ], int start, int end)
static void sort(char array[ ], int start, int end)
static void sort(double array[ ], int start, int end)
static void sort(float array[ ], int start, int end)
static void sort(int array[ ], int start, int end)
static void sort(long array[ ], int start, int end)
static void sort(short array[ ], int start, int end)
static void sort(Object array[ ], int start, int end)
static void sort(Object array[ ], int start, int end, Comparator c)
```

Here, the range beginning at *start* and running through *end*–1 within *array* will be sorted. In the last form, *c* is a **Comparator** that is used to order the elements of *array*

All of these methods can throw an **IllegalArgumentException** if *start* is greater than *end*, or an **ArrayIndexOutOfBoundsException** if *start* or *end* is out of bounds. The last two forms can also throw a **ClassCastException** if elements of the array being sorted are not comparable.

The following program illustrates how to use some of the methods of the **Arrays** class:

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

class ArraysDemo {
  public static void main(String args[]) {

    // allocate and initialize array
    int array[] = new int[10];
    for(int i = 0; i < 10; i++)
      array[i] = -3 * i;

    // display, sort, display
    System.out.print("Original contents: ");
    display(array);
    Arrays.sort(array);
    System.out.print("Sorted: ");
    display(array);

    // fill and display
    Arrays.fill(array, 2, 6, -1);
    System.out.print("After fill(): ");
    display(array);

    // sort and display
    Arrays.sort(array);
    System.out.print("After sorting again: ");
    display(array);

    // binary search for -9
    System.out.print("The value -9 is at location ");
    int index =
      Arrays.binarySearch(array, -9);
    System.out.println(index);
  }

  static void display(int array[]) {
    for(int i = 0; i < array.length; i++)
      System.out.print(array[i] + " ");
```

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

The following is the output from this program:

```
Original contents: 0 -3 -6 -9 -12 -15 -18 -21 -24 -27
Sorted: -27 -24 -21 -18 -15 -12 -9 -6 -3 0
After fill(): -27 -24 -1 -1 -1 -1 -9 -6 -3 0
After sorting again: -27 -24 -9 -6 -3 -1 -1 -1 -1 0
The value -9 is at location 2
```