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

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

    // Get and set a value.

    String val = ll.get(2);
    ll.set(2, 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(E)** append items to the end of the list, as do calls to **addLast( )**. To insert items at a specific location, use the **add(int, E)** 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. **HashSet** is a generic class that has this declaration:

```
class HashSet<E>
```

Here, **E** specifies the type of objects that the set will hold.

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 **add( )**, **contains( )**, **remove( )**, and **size( )** to remain constant even for large sets.

The following constructors are defined:

```
HashSet( )
HashSet(Collection<? extends E> 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 default capacity is 16.) 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.

It is important to note that **HashSet** 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<String> hs = new HashSet<String>();

    // Add elements to the hash set.
    hs.add("Beta");
    hs.add("Alpha");
    hs.add("Eta");
    hs.add("Gamma");
    hs.add("Epsilon");
    hs.add("Omega");

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

The following is the output from this program:

```
[Gamma, Eta, Alpha, Epsilon, Omega, Beta]
```

As explained, the elements are not stored in sorted order, and the precise output may vary.

The LinkedHashSet Class

The **LinkedHashSet** class extends **HashSet** and adds no members of its own. It is a generic class that has this declaration:

```
class LinkedHashSet<E>
```

Here, **E** specifies the type of objects that the set will hold. Its constructors parallel those in **HashSet**.

LinkedHashSet maintains a linked list of the entries in the set, in the order in which they were inserted. This allows insertion-order iteration over the set. That is, when cycling through a **LinkedHashSet** using an iterator, the elements will be returned in the order in which they were inserted. This is also the order in which they are contained in the string returned by **toString( )** when called on a **LinkedHashSet** object. To see the effect of **LinkedHashSet**, try substituting **LinkedHashSet** for **HashSet** in the preceding program. The output will be

```
[Beta, Alpha, Eta, Gamma, Epsilon, Omega]
```

which is the order in which the elements were inserted.

The TreeSet Class

TreeSet extends **AbstractSet** and implements the **NavigableSet** interface. It creates a collection that uses a tree for storage. Objects are stored in sorted, ascending order. Access and retrieval times are quite fast, which makes **TreeSet** an excellent choice when storing large amounts of sorted information that must be found quickly.

TreeSet is a generic class that has this declaration:

class TreeSet<E>

Here, **E** specifies the type of objects that the set will hold.

TreeSet has the following constructors:

TreeSet( )
TreeSet(Collection<? extends E> *c*)
TreeSet(Comparator<? super E> *comp*)
TreeSet(SortedSet<E> *ss*)

The first form constructs an empty tree set that will be sorted in ascending order according to the natural order of its elements. The second form builds a tree set that contains the elements of *c*. The third form constructs an empty tree set that will be sorted according to the comparator specified by *comp*. (Comparators are described later in this chapter.) The fourth form builds a tree set that contains the elements of *ss*.

Here is an example that demonstrates a **TreeSet**:

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

class TreeSetDemo {
  public static void main(String args[]) {
    // Create a tree set.
    TreeSet<String> ts = new TreeSet<String>();

    // Add elements to the tree set.
    ts.add("C");
    ts.add("A");
    ts.add("B");
    ts.add("E");
    ts.add("F");
    ts.add("D");
```

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

The output from this program is shown here:

```
[A, B, C, D, E, F]
```

As explained, because **TreeSet** stores its elements in a tree, they are automatically arranged in sorted order, as the output confirms.

Because **TreeSet** implements the **NavigableSet** interface, you can use the methods defined by **NavigableSet** to retrieve elements of a **TreeSet**. For example, assuming the preceding program, the following statement uses **subSet( )** to obtain a subset of **ts** that contains the elements between **C** (inclusive) and **F** (exclusive). It then displays the resulting set.

```
System.out.println(ts.subSet("C", "F"));
```

The output from this statement is shown here:

```
[C, D, E]
```

You might want to experiment with the other methods defined by **NavigableSet**.

The PriorityQueue Class

PriorityQueue extends **AbstractQueue** and implements the **Queue** interface. It creates a queue that is prioritized based on the queue's comparator. **PriorityQueue** is a generic class that has this declaration:

```
class PriorityQueue<E>
```

Here, **E** specifies the type of objects stored in the queue. **PriorityQueue**s are dynamic, growing as necessary.

PriorityQueue defines the six constructors shown here:

```
PriorityQueue( )
PriorityQueue(int capacity)
PriorityQueue(Comparator<? super E> comp) (Added by JDK 8.)
PriorityQueue(int capacity, Comparator<? super E> comp)
PriorityQueue(Collection<? extends E> c)
PriorityQueue(PriorityQueue<? extends E> c)
PriorityQueue(SortedSet<? extends E> c)
```

The first constructor builds an empty queue. Its starting capacity is 11. The second constructor builds a queue that has the specified initial capacity. The third constructor specifies a comparator, and the fourth builds a queue with the specified capacity and comparator. The last three constructors create queues that are initialized with the elements of the collection passed in *c*. In all cases, the capacity grows automatically as elements are added.

If no comparator is specified when a **PriorityQueue** is constructed, then the default comparator for the type of data stored in the queue is used. The default comparator will order the queue in ascending order. Thus, the head of the queue will be the smallest value. However, by providing a custom comparator, you can specify a different ordering scheme. For example, when storing items that include a time stamp, you could prioritize the queue such that the oldest items are first in the queue.

You can obtain a reference to the comparator used by a **PriorityQueue** by calling its **comparator( )** method, shown here:

Comparator<? super E> comparator( )

It returns the comparator. If natural ordering is used for the invoking queue, **null** is returned.

One word of caution: Although you can iterate through a **PriorityQueue** using an iterator, the order of that iteration is undefined. To properly use a **PriorityQueue**, you must call methods such as **offer( )** and **poll( )**, which are defined by the **Queue** interface.

The ArrayDeque Class

The **ArrayDeque** class extends **AbstractCollection** and implements the **Deque** interface. It adds no methods of its own. **ArrayDeque** creates a dynamic array and has no capacity restrictions. (The **Deque** interface supports implementations that restrict capacity, but does not require such restrictions.) **ArrayDeque** is a generic class that has this declaration:

class ArrayDeque<E>

Here, **E** specifies the type of objects stored in the collection.

ArrayDeque defines the following constructors:

ArrayDeque( )
ArrayDeque(int *size*)
ArrayDeque(Collection<? extends E> *c*)

The first constructor builds an empty deque. Its starting capacity is 16. The second constructor builds a deque that has the specified initial capacity. The third constructor creates a deque that is initialized with the elements of the collection passed in *c*. In all cases, the capacity grows as needed to handle the elements added to the deque.

The following program demonstrates **ArrayDeque** by using it to create a stack:

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

class ArrayDequeDemo {
  public static void main(String args[]) {
    // Create an array deque.
    ArrayDeque<String> adq = new ArrayDeque<String>();

    // Use an ArrayDeque like a stack.
    adq.push("A");
    adq.push("B");
    adq.push("D");
```

```
    adq.push("E");
    adq.push("F");

    System.out.print("Popping the stack: ");

    while(adq.peek() != null)
      System.out.print(adq.pop() + " ");

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

The output is shown here:

```
Popping the stack: F E D B A
```


The EnumSet Class

EnumSet extends **AbstractSet** and implements **Set**. It is specifically for use with elements of an **enum** type. It is a generic class that has this declaration:

```
class EnumSet<E extends Enum<E>>
```

Here, **E** specifies the elements. Notice that **E** must extend **Enum<E>**, which enforces the requirement that the elements must be of the specified **enum** type.

EnumSet defines no constructors. Instead, it uses the factory methods shown in Table 18-7 to create objects. All methods can throw **NullPointerException**. The **copyOf( )** and **range( )** methods can also throw **IllegalArgumentException**. Notice that the **of( )** method is overloaded a number of times. This is in the interest of efficiency. Passing a known number of arguments can be faster than using a vararg parameter when the number of arguments is small.

Accessing a Collection via an Iterator

Often, you will want to cycle through the elements in a collection. For example, you might want to display each element. One way to do this is to employ an *iterator*, which is an object that implements either the **Iterator** or the **ListIterator** interface. **Iterator** enables you to cycle through a collection, obtaining or removing elements. **ListIterator** extends **Iterator** to allow bidirectional traversal of a list, and the modification of elements. **Iterator** and **ListIterator** are generic interfaces which are declared as shown here:

```
interface Iterator<E>
interface ListIterator<E>
```

Here, **E** specifies the type of objects being iterated. The **Iterator** interface declares the methods shown in Table 18-8. The methods declared by **ListIterator** (along with those inherited from **Iterator**) are shown in Table 18-9. In both cases, operations that modify the underlying collection are optional. For example, **remove( )** will throw **UnsupportedOperationException** when used with a read-only collection. Various other exceptions are possible.

Method	Description
static <E extends Enum<E>> EnumSet<E> allOf(Class<E> *t*)	Creates an **EnumSet** that contains the elements in the enumeration specified by *t*.
static <E extends Enum<E>> EnumSet<E> complementOf(EnumSet<E> *e*)	Creates an **EnumSet** that is comprised of those elements not stored in *e*.
static <E extends Enum<E>> EnumSet<E> copyOf(EnumSet<E> *c*)	Creates an **EnumSet** from the elements stored in *c*.
static <E extends Enum<E>> EnumSet<E> copyOf(Collection<E> *c*)	Creates an **EnumSet** from the elements stored in *c*.
static <E extends Enum<E>> EnumSet<E> noneOf(Class<E> *t*)	Creates an **EnumSet** that contains the elements that are not in the enumeration specified by *t*, which is an empty set by definition.
static <E extends Enum<E>> EnumSet<E> of(E *v*, E … *varargs*)	Creates an **EnumSet** that contains *v* and zero or more additional enumeration values.
static <E extends Enum<E>> EnumSet<E> of(E *v*)	Creates an **EnumSet** that contains *v*.
static <E extends Enum<E>> EnumSet<E> of(E *v1*, E *v2*)	Creates an **EnumSet** that contains *v1* and *v2*.
static <E extends Enum<E>> EnumSet<E> of(E *v1*, E *v2*, E *v3*)	Creates an **EnumSet** that contains *v1* through *v3*.
static <E extends Enum<E>> EnumSet<E> of(E *v1*, E *v2*, E *v3*, E *v4*)	Creates an **EnumSet** that contains *v1* through *v4*.
static <E extends Enum<E>> EnumSet<E> of(E *v1*, E *v2*, E *v3*, E *v4*, E *v5*)	Creates an **EnumSet** that contains *v1* through *v5*.
static <E extends Enum<E>> EnumSet<E> range(E *start*, E *end*)	Creates an **EnumSet** that contains the elements in the range specified by *start* and *end*.

Table 18-7 The Methods Declared by **EnumSet**

Method	Description
default void forEachRemaining(Consumer<? super E> *action*)	The action specified by *action* is executed on each unprocessed element in the collection. (Added by JDK 8.)
boolean hasNext( )	Returns **true** if there are more elements. Otherwise, returns **false**.
E next( )	Returns the next element. Throws **NoSuchElementException** if there is not a next element.
default 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( )**. The default version throws an **UnsupportedOperationException**.

Table 18-8 The Methods Declared by **Iterator**

Method	Description
void add(E *obj*)	Inserts *obj* into the list in front of the element that will be returned by the next call to **next( )**.
default void forEachRemaining(Consumer<? super E> *action*)	The action specified by *action* is executed on each unprocessed element in the collection. (Added by JDK 8.)
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**.
E 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.
E 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(E *obj*)	Assigns *obj* to the current element. This is the element last returned by a call to either **next( )** or **previous( )**.

Table 18-9 The Methods Provided by **ListIterator**

NOTE Beginning with JDK 8, you can also use a **Spliterator** to cycle through a collection. **Spliterator** works differently than does **Iterator**, and it is described later in this chapter.

Using an Iterator

Before you can access a collection through an iterator, you must obtain one. 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( )**. 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**.

The following example implements these steps, demonstrating both the **Iterator** and **ListIterator** interfaces. 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<String> al = new ArrayList<String>();

    // 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<String> itr = al.iterator();
    while(itr.hasNext()) {
      String element = itr.next();
      System.out.print(element + " ");
    }
    System.out.println();

    // Modify objects being iterated.
    ListIterator<String> litr = al.listIterator();
    while(litr.hasNext()) {
      String element = litr.next();
      litr.set(element + "+");
    }

    System.out.print("Modified contents of al: ");
    itr = al.iterator();
    while(itr.hasNext()) {
      String element = itr.next();
      System.out.print(element + " ");
    }
    System.out.println();

    // Now, display the list backwards.
    System.out.print("Modified list backwards: ");
    while(litr.hasPrevious()) {
```

```
      String 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.

The For-Each Alternative to Iterators

If you won't be modifying the contents of a collection or obtaining elements in reverse order, then the for-each version of the **for** loop is often a more convenient alternative to cycling through a collection than is using an iterator. Recall that the **for** can cycle through any collection of objects that implement the **Iterable** interface. Because all of the collection classes implement this interface, they can all be operated upon by the **for**.

The following example uses a **for** loop to sum the contents of a collection:

```
// Use the for-each for loop to cycle through a collection.
import java.util.*;

class ForEachDemo {
  public static void main(String args[]) {
    // Create an array list for integers.
    ArrayList<Integer> vals = new ArrayList<Integer>();

    // Add values to the array list.
    vals.add(1);
    vals.add(2);
    vals.add(3);
    vals.add(4);
    vals.add(5);

    // Use for loop to display the values.
    System.out.print("Contents of vals: ");
    for(int v : vals)
      System.out.print(v + " ");

    System.out.println();

    // Now, sum the values by using a for loop.
```

```
    int sum = 0;
    for(int v : vals)
      sum += v;

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

The output from the program is shown here:

```
Contents of vals: 1 2 3 4 5
Sum of values: 15
```

As you can see, the **for** loop is substantially shorter and simpler to use than the iterator-based approach. However, it can only be used to cycle through a collection in the forward direction, and you can't modify the contents of the collection.

Spliterators

JDK 8 adds a new type of iterator called a *spliterator* that is defined by the **Spliterator** interface. A spliterator cycles through a sequence of elements, and in this regard, it is similar to the iterators just described. However, the techniques required to use it differ. Furthermore, it offers substantially more functionality than does either **Iterator** or **ListIterator**. Perhaps the most important aspect of **Spliterator** is its ability to provide support for parallel iteration of portions of the sequence. Thus, **Spliterator** supports parallel programming. (See Chapter 28 for information on concurrency and parallel programming.) However, you can use **Spliterator** even if you won't be using parallel execution. One reason you might want to do so is because it offers a streamlined approach that combines the *hasNext* and *next* operations into one method.

Spliterator is a generic interface that is declared like this:

```
interface Spliterator<T>
```

Here, **T** is the type of elements being iterated. **Spliterator** declares the methods shown in Table 18-10.

Using **Spliterator** for basic iteration tasks is quite easy: simply call **tryAdvance( )** until it returns **false**. If you will be applying the same action to each element in the sequence, **forEachRemaining( )** offers a streamlined alternative. In both cases, the action that will occur with each iteration is defined by what the **Consumer** object does with each element. **Consumer** is a functional interface that applies an action to an object. It is a generic functional interface declared in **java.util.function**. (See Chapter 19 for information on **java.util.function**.) **Consumer** specifies only one abstract method, **accept( )**, which is shown here:

```
void accept(T objRef)
```

In the case of **tryAdvance( )**, each iteration passes the next element in the sequence to *objRef*. Often, the easiest way to implement **Consumer** is by use of a lambda expression.

Method	Description
int characteristics( )	Returns the characteristics of the invoking spliterator, encoded into an integer.
long estimateSize( )	Estimates the number of elements left to iterate and returns the result. Returns **Long.MAX_VALUE** if the count cannot be obtained for any reason.
default void forEachRemaining(Consumer<? super T> *action*)	Applies *action* to each unprocessed element in the data source.
default Comparator<? super T> getComparator( )	Returns the comparator used by the invoking spliterator or **null** if natural ordering is used. If the sequence is unordered, **IllegalStateException** is thrown.
default long getExactSizeIfKnown( )	If the invoking spliterator is sized, returns the number of elements left to iterate. Returns –1 otherwise.
default boolean hasCharacteristics(int *val*)	Returns **true** if the invoking spliterator has the characteristics passed in *val*. Returns **false** otherwise.
boolean tryAdvance(Consumer<? super T> *action*)	Executes *action* on the next element in the iteration. Returns **true** if there is a next element. Returns **false** if no elements remain.
Spliterator<T> trySplit( )	If possible, splits the invoking spliterator, returning a reference to a new spliterator for the partition. Otherwise, returns **null**. Thus, if successful, the original spliterator iterates over one portion of the sequence and the returned spliterator iterates over the other portion.

Table 18-10 The Methods Declared by **Spliterator**

The following program provides a simple example of **Spliterator**. Notice that the program demonstrates both **tryAdvance( )** and **forEachRemaining( )**. Also notice how these methods combine the actions of **Iterator**'s **next( )** and **hasNext( )** methods into a single call.

```
// A simple Spliterator demonstration.
import java.util.*;

class SpliteratorDemo {

  public static void main(String args[]) {
    // Create an array list for doubles.
    ArrayList<Double> vals = new ArrayList<>();

    // Add values to the array list.
    vals.add(1.0);
    vals.add(2.0);
    vals.add(3.0);
    vals.add(4.0);
    vals.add(5.0);
```

```
    // Use tryAdvance() to display contents of vals.
    System.out.print("Contents of vals:\n");
    Spliterator<Double> spltitr = vals.spliterator();
    while(spltitr.tryAdvance((n) -> System.out.println(n)));
    System.out.println();

    // Create new list that contains square roots.
    spltitr = vals.spliterator();
    ArrayList<Double> sqrs = new ArrayList<>();
    while(spltitr.tryAdvance((n) -> sqrs.add(Math.sqrt(n))));

    // Use forEachRemaining() to display contents of sqrs.
    System.out.print("Contents of sqrs:\n");
    spltitr = sqrs.spliterator();
    spltitr.forEachRemaining((n) -> System.out.println(n));
    System.out.println();
  }
}
```

The output is shown here:

```
Contents of vals:
1.0
2.0
3.0
4.0
5.0

Contents of sqrs:
1.0
1.4142135623730951
1.7320508075688772
2.0
2.23606797749979
```

Although this program demonstrates the mechanics of using **Spliterator**, it does not reveal its full power. As mentioned, **Spliterator**'s maximum benefit is found in situations that involve parallel processing.

In Table 18-10, notice the methods **characteristics( )** and **hasCharacteristics( )**. Each **Spliterator** has a set of attributes, called *characteristics*, associated with it. These are defined by static **int** fields in **Spliterator**, such as **SORTED**, **DISTINCT**, **SIZED**, and **IMMUTABLE**, to name a few. You can obtain the characteristics by calling **characteristics( )**. You can determine if a characteristic is present by calling **hasCharacteristics( )**. Often, you won't need to access a **Spliterator**'s characteristics, but in some cases, they can aid in creating efficient, resilient code.

NOTE For a further discussion of **Spliterator**, see Chapter 29, where it is used in the context of the new stream API. For a discussion of lambda expressions, see Chapter 15. See Chapter 28 for a discussion of parallel programming and concurrency.

There are several nested subinterfaces of **Spliterator** designed for use with the primitive types **double**, **int**, and **long**. These are called **Spliterator.OfDouble**, **Spliterator.OfInt**, and **Spliterator.OfLong**. There is also a generalized version called **Spliterator.OfPrimitive( )**, which offers additional flexibility and serves as a superinterface of the aforementioned ones.

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<Address> ml = new LinkedList<Address>();

    // 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"));
```

```
    // Display the mailing list.
    for(Address element : ml)
      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

The **RandomAccess** 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, among others.

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.

There is one key point about maps that is important to mention at the outset: they don't implement the **Iterable** interface. This means that you *cannot* cycle through a map using a for-each style **for** loop. Furthermore, you can't obtain an iterator to a map. However, as you will soon see, you can obtain a collection-view of a map, which does allow the use of either the **for** loop or an iterator.

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**.
NavigableMap	Extends **SortedMap** to handle the retrieval of entries based on closest-match searches.
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. **Map** is generic and is declared as shown here:

```
interface Map<K, V>
```

Here, **K** specifies the type of keys, and **V** specifies the type of values.

The methods declared by **Map** are summarized in Table 18-11. Several methods throw a **ClassCastException** 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. An **IllegalArgumentException** is thrown if an invalid argument is used.

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, although part of the Collections Framework, maps are not, themselves, collections because they do not implement the **Collection** interface. However, 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( )**. For all three collection-views, the collection is backed by the map. Changing one affects the other. Collection-views are the means by which maps are integrated into the larger Collections Framework.

Method	Description
void clear( )	Removes all key/value pairs from the invoking map.
default V compute(K *k*, BiFunction<? super K, ? super V, ? extends V> *func*)	Calls *func* to construct a new value. If func returns non-**null**, the new key/value pair is added to the map, any preexisting pairing is removed, and the new value is returned. If *func* returns **null**, any preexisting pairing is removed, and **null** is returned. (Added by JDK 8.)
default V computeIfAbsent(K *k*, Function<? super K, ? extends V> *func*)	Returns the value associated with the key *k*. Otherwise, the value is constructed through a call to *func* and the pairing is entered into the map and the constructed value is returned. If no value can be constructed, **null** is returned. (Added by JDK 8.)
default V computeIfPresent(K *k*, BiFunction<? super K, ? super V, ? extends V> *func*)	If *k* is in the map, a new value is constructed through a call to *func* and the new value replaces the old value in the map. In this case, the new value is returned. If the value returned by *func* is **null**, the existing key and value are removed from the map and **null** is returned. (Added by JDK 8.)
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<Map.Entry<K, V>> entrySet( )	Returns a **Set** that contains the entries in the map. The set contains objects of type **Map.Entry**. Thus, 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**.
default void forEach(BiConsumer< ? super K, ? super V> *action*)	Executes *action* on each element in the invoking map. A **ConcurrentModificationException** will be thrown if an element is removed during the process. (Added by JDK 8.)
V get(Object *k*)	Returns the value associated with the key *k*. Returns **null** if the key is not found.
default V getOrDefault(Object *k*, V *defVal*)	Returns the value associated with *k* if it is in the map. Otherwise, *defVal* is returned. (Added by JDK 8.)
int hashCode( )	Returns the hash code for the invoking map.
boolean isEmpty( )	Returns **true** if the invoking map is empty. Otherwise, returns **false**.
Set<K> 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.

Table 18-11 The Methods Declared by **Map**

Method	Description
default V merge(K *k*, V *v*, BiFunction<? super V, ? super V, ? extends V> *func*)	If *k* is not in the map, the pairing *k*,*v* is added to the map. In this case, *v* is returned. Otherwise, *func* returns a new value based on the old value, the key is updated to use this value, and **merge( )** returns this value. If the value returned by *func* is **null**, the existing key and value are removed from the map and **null** is returned. (Added by JDK 8.)
V put(K *k*, V *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<? extends K, ? extends V> *m*)	Puts all the entries from *m* into this map.
default V putIfAbsent(K *k*, V *v*)	Inserts the key/value pair into the invoking map if this pairing is not already present or if the existing value is **null**. Returns the old value. The **null** value is returned when no previous mapping exists, or the value is **null**. (Added by JDK 8.)
V remove(Object *k*)	Removes the entry whose key equals *k*.
default boolean remove(Object *k*, Object *v*)	If the key/value pair specified by *k* and *v* is in the invoking map, it is removed and **true** is returned. Otherwise, **false** is returned. (Added by JDK 8.)
default boolean replace(K *k*, V *oldV*, V *newV*)	If the key/value pair specified by *k* and *oldV* is in the invoking map, the value is replaced by *newV* and **true** is returned. Otherwise **false** is returned. (Added by JDK 8.)
default V replace(K *k*, V *v*)	If the key specified by *k* is in the invoking map, its value is set to *v* and the previous value is returned. Otherwise, **null** is returned. (Added by JDK 8.)
default void replaceAll(BiFunction< ? super K, ? super V, ? extends V> *func*)	Executes *func* on each element of the invoking map, replacing the element with the result returned by *func*. A **ConcurrentModificationException** will be thrown if an element is removed during the process. (Added by JDK 8.)
int size( )	Returns the number of key/value pairs in the map.
Collection<V> values( )	Returns a collection containing the values in the map. This method provides a collection-view of the values in the map.

Table 18-11 The Methods Declared by **Map** *(continued)*

The SortedMap Interface

The **SortedMap** interface extends **Map**. It ensures that the entries are maintained in ascending order based on the keys. **SortedMap** is generic and is declared as shown here:

```
interface SortedMap<K, V>
```

Here, **K** specifies the type of keys, and **V** specifies the type of values.

The methods declared by **SortedMap** are summarized in Table 18-12. 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. An **IllegalArgumentException** is thrown if an invalid argument is used.

Sorted maps allow very efficient manipulations of *submaps* (in other words, subsets of a map). To obtain a submap, use **headMap( )**, **tailMap( )**, or **subMap( )**. The submap returned by these methods is backed by the invoking map. Changing one changes the other. To get the first key in the set, call **firstKey( )**. To get the last key, use **lastKey( )**.

The NavigableMap Interface

The **NavigableMap** interface extends **SortedMap** and declares the behavior of a map that supports the retrieval of entries based on the closest match to a given key or keys. **NavigableMap** is a generic interface that has this declaration:

```
interface NavigableMap<K,V>
```

Here, **K** specifies the type of the keys, and **V** specifies the type of the values associated with the keys. In addition to the methods that it inherits from **SortedMap**, **NavigableMap** adds those summarized in Table 18-13. Several methods throw a **ClassCastException** when an object is incompatible with the keys in the map. A **NullPointerException** is thrown if an attempt is made to use a **null** object and **null** keys are not allowed in the set. An **IllegalArgumentException** is thrown if an invalid argument is used.

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

Table 18-12 The Methods Declared by **SortedMap**

Method	Description
Map.Entry<K,V> ceilingEntry(K *obj*)	Searches the map for the smallest key *k* such that *k* >= *obj*. If such a key is found, its entry is returned. Otherwise, **null** is returned.
K ceilingKey(K *obj*)	Searches the map for the smallest key *k* such that *k* >= *obj*. If such a key is found, it is returned. Otherwise, **null** is returned.
NavigableSet<K> descendingKeySet( )	Returns a **NavigableSet** that contains the keys in the invoking map in reverse order. Thus, it returns a reverse set-view of the keys. The resulting set is backed by the map.
NavigableMap<K,V> descendingMap( )	Returns a **NavigableMap** that is the reverse of the invoking map. The resulting map is backed by the invoking map.
Map.Entry<K,V> firstEntry( )	Returns the first entry in the map. This is the entry with the least key.
Map.Entry<K,V> floorEntry(K *obj*)	Searches the map for the largest key *k* such that *k* <= *obj*. If such a key is found, its entry is returned. Otherwise, **null** is returned.
K floorKey(K *obj*)	Searches the map for the largest key *k* such that *k* <= *obj*. If such a key is found, it is returned. Otherwise, **null** is returned.
NavigableMap<K,V> headMap(K *upperBound*, boolean *incl*)	Returns a **NavigableMap** that includes all entries from the invoking map that have keys that are less than *upperBound*. If *incl* is **true**, then an element equal to *upperBound* is included. The resulting map is backed by the invoking map.
Map.Entry<K,V> higherEntry(K *obj*)	Searches the set for the largest key *k* such that *k* > *obj*. If such a key is found, its entry is returned. Otherwise, **null** is returned.
K higherKey(K *obj*)	Searches the set for the largest key *k* such that *k* > *obj*. If such a key is found, it is returned. Otherwise, **null** is returned.
Map.Entry<K,V> lastEntry( )	Returns the last entry in the map. This is the entry with the largest key.
Map.Entry<K,V> lowerEntry(K obj)	Searches the set for the largest key *k* such that *k* < *obj*. If such a key is found, its entry is returned. Otherwise, **null** is returned.
K lowerKey(K *obj*)	Searches the set for the largest key *k* such that *k* < *obj*. If such a key is found, it is returned. Otherwise, **null** is returned.

Table 18-13 The Methods Declared by **NavigableMap**

Method	Description
NavigableSet<K> navigableKeySet( )	Returns a **NavigableSet** that contains the keys in the invoking map. The resulting set is backed by the invoking map.
Map.Entry<K,V> pollFirstEntry( )	Returns the first entry, removing the entry in the process. Because the map is sorted, this is the entry with the least key value. **null** is returned if the map is empty.
Map.Entry<K,V> pollLastEntry( )	Returns the last entry, removing the entry in the process. Because the map is sorted, this is the entry with the greatest key value. **null** is returned if the map is empty.
NavigableMap<K,V> subMap(K *lowerBound*, boolean *lowIncl*, K *upperBound* boolean *highIncl*)	Returns a **NavigableMap** that includes all entries from the invoking map that have keys that are greater than *lowerBound* and less than *upperBound*. If *lowIncl* is **true**, then an element equal to *lowerBound* is included. If *highIncl* is **true**, then an element equal to *highIncl* is included. The resulting map is backed by the invoking map.
NavigableMap<K,V> tailMap(K *lowerBound*, boolean *incl*)	Returns a **NavigableMap** that includes all entries from the invoking map that have keys that are greater than *lowerBound*. If *incl* is **true**, then an element equal to *lowerBound* is included. The resulting map is backed by the invoking map.

Table 18-13 The Methods Declared by **NavigableMap** *(continued)*

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. **Map.Entry** is generic and is declared like this:

```
interface Map.Entry<K, V>
```

Here, **K** specifies the type of keys, and **V** specifies the type of values. Table 18-14 summarizes the non-static methods declared by **Map.Entry**. JDK 8 adds two static methods. The first is **comparingByKey( )**, which returns a **Comparator** that compares entries by key. The second is **comparingByValue( )**, which returns a **Comparator** that compares entries by value.

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.
K getKey( )	Returns the key for this map entry.
V getValue( )	Returns the value for this map entry.
int hashCode( )	Returns the hash code for this map entry.
V setValue(V *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 18-14 The Non-Static Methods Declared by **Map.Entry**

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.
EnumMap	Extends **AbstractMap** for use with **enum** keys.
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 otherwise unused. This class is not discussed further here. The other map classes are described next.

The HashMap Class

The **HashMap** class extends **AbstractMap** and implements the **Map** interface. It uses a hash table to store the map. This allows the execution time of **get( )** and **put( )** to remain constant even for large sets. **HashMap** is a generic class that has this declaration:

```
class HashMap<K, V>
```

Here, **K** specifies the type of keys, and **V** specifies the type of values.

The following constructors are defined:

HashMap( )
HashMap(Map<? extends K, ? extends V> *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. The default capacity is 16. The default fill ratio is 0.75.

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<String, Double> hm = new HashMap<String, Double>();

    // 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("Tod Hall", new Double(99.22));
    hm.put("Ralph Smith", new Double(-19.08));

    // Get a set of the entries.
    Set<Map.Entry<String, Double>> set = hm.entrySet();

    // Display the set.
    for(Map.Entry<String, Double> me : set) {
      System.out.print(me.getKey() + ": ");
      System.out.println(me.getValue());
    }

    System.out.println();

    // Deposit 1000 into John Doe's account.
    double balance = hm.get("John Doe");
    hm.put("John Doe", 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):

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

John Doe's new 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 extends **AbstractMap** and implements the **NavigableMap** interface. It creates maps stored in a tree structure. 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. **TreeMap** is a generic class that has this declaration:

```
class TreeMap<K, V>
```

Here, **K** specifies the type of keys, and **V** specifies the type of values.

The following **TreeMap** constructors are defined:

```
TreeMap( )
TreeMap(Comparator<? super K> comp)
TreeMap(Map<? extends K, ? extends V> m)
TreeMap(SortedMap<K, ? extends V> 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 has no map methods beyond those specified by the **NavigableMap** interface and the **AbstractMap** class.

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<String, Double> tm = new TreeMap<String, Double>();

    // 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("Tod Hall", new Double(99.22));
    tm.put("Ralph Smith", new Double(-19.08));

    // Get a set of the entries.
    Set<Map.Entry<String, Double>> set = tm.entrySet();

    // Display the elements.
    for(Map.Entry<String, Double> me : set) {
       System.out.print(me.getKey() + ": ");
       System.out.println(me.getValue());
    }
    System.out.println();

    // Deposit 1000 into John Doe's account.
    double balance = tm.get("John Doe");
    tm.put("John Doe", 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

LinkedHashMap extends **HashMap**. It 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 through a collection-view of 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** is a generic class that has this declaration:

```
class LinkedHashMap<K, V>
```

Here, **K** specifies the type of keys, and **V** specifies the type of values.

LinkedHashMap defines the following constructors:

```
LinkedHashMap( )
LinkedHashMap(Map<? extends K, ? extends V> 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 default capacity is 16. The default ratio is 0.75. 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<K, V> 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

IdentityHashMap extends **AbstractMap** and implements the **Map** interface. It is similar to **HashMap** except that it uses reference equality when comparing elements. **IdentityHashMap** is a generic class that has this declaration:

```
class IdentityHashMap<K, V>
```

Here, **K** specifies the type of key, and **V** specifies the type of value. The API documentation explicitly states that **IdentityHashMap** is not for general use.

The EnumMap Class

EnumMap extends **AbstractMap** and implements **Map**. It is specifically for use with keys of an **enum** type. It is a generic class that has this declaration:

```
class EnumMap<K extends Enum<K>, V>
```

Here, **K** specifies the type of key, and **V** specifies the type of value. Notice that **K** must extend **Enum<K>**, which enforces the requirement that the keys must be of an **enum** type.

EnumMap defines the following constructors:

EnumMap(Class<K> *kType*)
EnumMap(Map<K, ? extends V> *m*)
EnumMap(EnumMap<K, ? extends V> *em*)

The first constructor creates an empty **EnumMap** of type *kType*. The second creates an **EnumMap** map that contains the same entries as *m*. The third creates an **EnumMap** initialized with the values in *em*.

EnumMap defines no methods of its own.

Comparators

Both **TreeSet** and **TreeMap** store elements in sorted order. However, it is the comparator that defines precisely what "sorted order" means. By default, these classes store their elements by using what Java refers to as "natural ordering," which is usually the ordering that you would expect (A before B, 1 before 2, and so forth). If you want to order elements a different way, then specify a **Comparator** when you construct the set or map. Doing so gives you the ability to govern precisely how elements are stored within sorted collections and maps.

Comparator is a generic interface that has this declaration:

interface Comparator<T>

Here, **T** specifies the type of objects being compared.

Prior to JDK 8, the **Comparator** interface defined only two methods: **compare( )** and **equals( )**. The **compare( )** method, shown here, compares two elements for order:

int compare(T *obj1*, T *obj2*)

obj1 and *obj2* are the objects to be compared. Normally, this method returns zero if the objects are equal. It returns a positive value if *obj1* is greater than *obj2*. Otherwise, a negative value is returned. The method can throw a **ClassCastException** if the types of the objects are not compatible for comparison. By implementing **compare( )**, you can alter the way that objects are ordered. For example, to sort in reverse order, you can create a comparator that reverses the outcome of a comparison.

The **equals( )** method, shown here, tests whether an object equals the invoking comparator:

boolean equals(object *obj*)

Here, *obj* is the object to be tested for equality. The method returns **true** if *obj* and the invoking object are both **Comparator** objects and use the same ordering. Otherwise, it returns **false**. Overriding **equals( )** is not necessary, and most simple comparators will not do so.

For many years, the preceding two methods were the only methods defined by **Comparator**. With the release of JDK 8, the situation has dramatically changed. JDK 8 adds significant new functionality to **Comparator** through the use of default and static interface methods. Each is described here.

You can obtain a comparator that reverses the ordering of the comparator on which it is called by using **reversed( )**, shown here:

```
default Comparator<T> reversed( )
```

It returns the reverse comparator. For example, assuming a comparator that uses natural ordering for the characters A through Z, a reverse order comparator would put B before A, C before B, and so on.

A method related to **reversed( )** is **reverseOrder( )**, shown next:

```
static <T extends Comparable<? super T>> Comparator<T> reverseOrder( )
```

It returns a comparator that reverses the natural order of the elements. Conversely, you can obtain a comparator that uses natural ordering by calling the static method **naturalOrder( )**, shown next:

```
static <T extends Comparable<? super T>> Comparator<T> naturalOrder( )
```

If you want a comparator that can handle **null** values, use **nullsFirst( )** or **nullsLast( )**, shown here:

```
static <T> Comparator<T> nullsFirst(Comparator<? super T> comp)
static <T> Comparator<T> nullsLast(Comparator<? super T> comp)
```

The **nullsFirst( )** method returns a comparator that views **null** values as less than other values. The **nullsLast( )** method returns a comparator that views **null** values as greater than other values. In both cases, if the two values being compared are non-**null**, *comp* performs the comparison. If *comp* is passed **null**, then all non-**null** values are viewed as equivalent.

Another default method added by JDK 8 is **thenComparing( )**. It returns a comparator that performs a second comparison when the outcome of the first comparison indicates that the objects being compared are equal. Thus, it can be used to create a "compare by X then compare by Y" sequence. For example, when comparing cities, the first comparison might compare names, with the second comparison comparing states. (Therefore, Springfield, Illinois, would come before Springfield, Missouri, assuming normal, alphabetical order.) The **thenComparing( )** method has three forms. The first, shown here, lets you specify the second comparator by passing an instance of **Comparator**:

```
default Comparator<T> thenComparing(Comparator<? super T> thenByComp)
```

Here, *thenByComp* specifies the comparator that is called if the first comparison returns equal.

The next versions of **thenComparing( )** let you specify the standard functional interface **Function** (defined by **java.util.function**). They are shown here:

```
default <U extends Comparable<? super U> Comparator<T>
    thenComparing(Function<? super T, ? extends U> getKey)

default <U> Comparator<T>
    thenComparing(Function<? super T, ? extends U> getKey,
                  Comparator<? super U> keyComp)
```

In both, *getKey* refers to function that obtains the next comparison key, which is used if the first comparison returns equal. In the second version, *keyComp* specifies the comparator used to compare keys. (Here, and in subsequent uses, **U** specifies the type of the key.)

Comparator also adds the following specialized versions of "then comparing" methods for the primitive types:

```
default Comparator<T>
    thenComparingDouble(ToDoubleFunction<? super T> getKey)

default Comparator<T>
    thenComparingInt(ToIntFunction<? super T> getKey)

default Comparator<T>
    thenComparingLong(ToLongFunction<? super T> getKey)
```

In all methods, *getKey* refers to a function that obtains the next comparison key.

Finally, JDK 8 adds to **Comparator** a method called **comparing( )**. It returns a comparator that obtains its comparison key from a function passed to the method. There are two versions of **comparing( )**, shown here:

```
static <T, U extends Comparable<? super U>> Comparator<T>
    comparing(Function<? super T, ? extends U> getKey)

static <T, U> Comparator<T>
    comparing(Function<? super T, ? extends U> getKey,
              Comparator<? super U> keyComp)
```

In both, *getKey* refers to a function that obtains the next comparison key. In the second version, *keyComp* specifies the comparator used to compare keys. **Comparator** also adds the following specialized versions of these methods for the primitive types:

```
static <T> Comparator<T>
    ComparingDouble(ToDoubleFunction<? super T> getKey)

static <T> Comparator<T>
    ComparingInt(ToIntFunction<? super T> getKey)

static <T> Comparator<T>
    ComparingLong(ToLongFunction<? super T> getKey)
```

In all methods, *getKey* refers to a function that obtains the next comparison key.

Using a Comparator

The following is an example that demonstrates the power of a custom comparator. It implements the **compare( )** method for strings that operates in reverse of normal. Thus, it causes a tree set to be sorted in reverse order.

```
// Use a custom comparator.
import java.util.*;

// A reverse comparator for strings.
class MyComp implements Comparator<String> {
```

```
  public int compare(String aStr, String bStr) {

    // Reverse the comparison.
    return bStr.compareTo(aStr);
  }

  // No need to override equals or the default methods.
}

class CompDemo {
  public static void main(String args[]) {
    // Create a tree set.
    TreeSet<String> ts = new TreeSet<String>(new MyComp());

    // Add elements to the tree set.
    ts.add("C");
    ts.add("A");
    ts.add("B");
    ts.add("E");
    ts.add("F");
    ts.add("D");

    // Display the elements.
    for(String element : ts)
      System.out.print(element + " ");

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


As the following output shows, the tree is now sorted in reverse order:

```
F E D C B A
```

Look closely at the **MyComp** class, which implements **Comparator** by implementing **compare( )**. (As explained earlier, overriding **equals( )** is neither necessary nor common. It is also not necessary to override the default methods added by JDK 8.) Inside **compare( )**, the **String** method **compareTo( )** compares the two strings. However, **bStr**—not **aStr**—invokes **compareTo( )**. This causes the outcome of the comparison to be reversed.

Although the way in which the reverse order comparator is implemented by the preceding program is perfectly adequate, beginning with JDK 8, there is another way to approach a solution. It is now possible to simply call **reversed( )** on a natural-order comparator. It will return an equivalent comparator, except that it runs in reverse. For example, assuming the preceding program, you can rewrite **MyComp** as a natural-order comparator, as shown here:

```
class MyComp implements Comparator<String> {
  public int compare(String aStr, String bStr) {
    return aStr.compareTo(bStr);
  }
}
```

Next, you can use the following sequence to create a **TreeSet** that orders its string elements in reverse:

```
MyComp mc = new MyComp(); // Create a comparator

// Pass a reverse order version of MyComp to TreeSet.
TreeSet<String> ts = new TreeSet<String>(mc.reversed());
```

If you plug this new code into the preceding program, it will produce the same results as before. In this case, there is no advantage gained by using **reversed( )**. However, in cases in which you need to create both a natural-order comparator and a reversed comparator, then using **reversed( )** gives you an easy way to obtain the reverse-order comparator without having to code it explicitly.

Beginning with JDK 8, it is not actually necessary to create the **MyComp** class in the preceding examples because a lambda expression can be easily used instead. For example, you can remove the **MyComp** class entirely and create the string comparator by using this statement:

```
// Use a lambda expression to implement Comparator<String>.
Comparator<String> mc = (aStr, bStr) -> aStr.compareTo(bStr);
```

One other point: in this simple example, it would also be possible to specify a reverse comparator via a lambda expression directly in the call to the **TreeSet( )** constructor, as shown here:

```
// Pass a reversed comparator to TreeSet() via a
// lambda expression.
TreeSet<String> ts = new TreeSet<String>(
                          (aStr, bStr) -> bStr.compareTo(aStr));
```

By making these changes, the program is substantially shortened, as its final version shown here illustrates:

```
// Use a lambda expression to create a reverse comparator.
import java.util.*;

class CompDemo2 {
  public static void main(String args[]) {

    // Pass a reverse comparator to TreeSet() via a
    // lambda expression.
    TreeSet<String> ts = new TreeSet<String>(
                               (aStr, bStr) -> bStr.compareTo(aStr));

    // Add elements to the tree set.
    ts.add("C");
    ts.add("A");
    ts.add("B");
    ts.add("E");
    ts.add("F");
    ts.add("D");
```

```
    // Display the elements.
    for(String element : ts)
      System.out.print(element + " ");

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

For a more practical example that uses a custom comparator, the following program is an updated version of the **TreeMap** program shown earlier that stores account balances. In the previous version, the accounts were sorted by name, but the sorting began with the first name. The following program sorts the accounts by last name. To do so, it uses a comparator that compares the last name of each account. This results in the map being sorted by last name.


```
// Use a comparator to sort accounts by last name.
import java.util.*;

// Compare last whole words in two strings.
class TComp implements Comparator<String> {
  public int compare(String aStr, String bStr) {
    int i, j, k;

    // Find index of beginning of last name.
    i = aStr.lastIndexOf(' ');
    j = bStr.lastIndexOf(' ');

    k = aStr.substring(i).compareToIgnoreCase (bStr.substring(j));
    if(k==0) // last names match, check entire name
      return aStr.compareToIgnoreCase (bStr);
    else
      return k;
  }

  // No need to override equals.
}

class TreeMapDemo2 {
  public static void main(String args[]) {
    // Create a tree map.
    TreeMap<String, Double> tm = new TreeMap<String, Double>(new TComp());

    // 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("Tod Hall", new Double(99.22));
    tm.put("Ralph Smith", new Double(-19.08));

    // Get a set of the entries.
    Set<Map.Entry<String, Double>> set = tm.entrySet();
```

```
    // Display the elements.
    for(Map.Entry<String, Double> me : set) {
      System.out.print(me.getKey() + ": ");
      System.out.println(me.getValue());
    }
    System.out.println();

    // Deposit 1000 into John Doe's account.
    double balance =  tm.get("John Doe");
    tm.put("John Doe", balance + 1000);

    System.out.println("John Doe's new balance: " +
      tm.get("John Doe"));
  }
}
```

Here is the output; notice that the accounts are now sorted by last name:

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

John Doe's new balance: 4434.34
```

The comparator class **TComp** compares two strings that hold first and last names. It does so by first comparing last names. To do this, it finds the index of the last space in each string and then compares the substrings of each element that begin at that point. In cases where last names are equivalent, the first names are then compared. This yields a tree map that is sorted by last name, and within last name by first name. You can see this because Ralph Smith comes before Tom Smith in the output.

If you are using JDK 8 or later, then there is another way that you could code the preceding program so the map is sorted by last name and then by first name. This approach uses the **thenComparing( )** method. Recall that **thenComparing( )** lets you specify a second comparator that will be used if the invoking comparator returns equal. This approach is put into action by the following program, which reworks the preceding example to use **thenComparing( )**:

```
// Use thenComparing() to sort by last, then first name.
import java.util.*;

// A comparator that compares last names.
class CompLastNames implements Comparator<String> {
  public int compare(String aStr, String bStr) {
    int i, j;

    // Find index of beginning of last name.
    i = aStr.lastIndexOf(' ');
    j = bStr.lastIndexOf(' ');
```

```
    return aStr.substring(i).compareToIgnoreCase(bStr.substring(j));
  }
}

// Sort by entire name when last names are equal.
class CompThenByFirstName implements Comparator<String> {
  public int compare(String aStr, String bStr) {
    int i, j;

    return aStr.compareToIgnoreCase(bStr);
  }
}

class TreeMapDemo2A {
  public static void main(String args[]) {
    // Use thenComparing() to create a comparator that compares
    // last names, then compares entire name when last names match.
    CompLastNames compLN = new CompLastNames();
    Comparator<String> compLastThenFirst =
                         compLN.thenComparing(new CompThenByFirstName());

    // Create a tree map.
    TreeMap<String, Double> tm =
                         new TreeMap<String, Double>(compLastThenFirst);

    // 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("Tod Hall", new Double(99.22));
    tm.put("Ralph Smith", new Double(-19.08));

    // Get a set of the entries.
    Set<Map.Entry<String, Double>> set = tm.entrySet();

    // Display the elements.
    for(Map.Entry<String, Double> me : set) {
      System.out.print(me.getKey() + ": ");
      System.out.println(me.getValue());
    }
    System.out.println();

    // Deposit 1000 into John Doe's account.
    double balance =  tm.get("John Doe");
    tm.put("John Doe", balance + 1000);

    System.out.println("John Doe's new balance: " +
      tm.get("John Doe"));
  }
}
```

This version produces the same output as before. It differs only in how it accomplishes its job. To begin, notice that a comparator called **CompLastNames** is created. This comparator

compares only the last names. A second comparator, called **CompThenByFirstName**, compares the entire name, starting with the first name. Next, the **TreeMap** is then created by the following sequence:

```
CompLastNames compLN = new CompLastNames();
Comparator<String> compLastThenFirst =
                    compLN.thenComparing(new CompThenByFirstName());
```

Here, the primary comparator is **compLN**. It is an instance of **CompLastNames**. On it is called **thenComparing( )**, passing in an instance of **CompThenByFirstName**. The result is assigned to the comparator called **compLastThenFirst**. This comparator is used to construct the **TreeMap**, as shown here:

```
TreeMap<String, Double> tm =
                    new TreeMap<String, Double>(compLastThenFirst);
```

Now, whenever the last names of the items being compared are equal, the entire name, beginning with the first name, is used to order the two. This means that names are ordered based on last name, and within last names, by first names.

One last point: in the interest of clarity, this example explicitly creates two comparator classes called **CompLastNames** and **ThenByFirstNames**, but lambda expressions could have been used instead. You might want to try this on your own. Just follow the same general approach described for the **CompDemo2** example shown earlier.

The Collection Algorithms

The Collections Framework defines several algorithms that can be applied to collections and maps. These algorithms are defined as static methods within the **Collections** class. They are summarized in Table 18-15. As explained earlier, beginning with JDK 5 all of the algorithms were retrofitted for generics.

Several of the methods can throw a **ClassCastException**, which occurs when an attempt is made to compare incompatible types, or an **UnsupportedOperationException**, which occurs when an attempt is made to modify an unmodifiable collection. Other exceptions are possible, depending on the method.

One thing to pay special attention to is the set of **checked** methods, such as **checkedCollection( )**, which returns what the API documentation refers to as a "dynamically typesafe view" of a collection. This view is a reference to the collection that monitors insertions into the collection for type compatibility at run time. An attempt to insert an incompatible element will cause a **ClassCastException**. Using such a view is especially helpful during debugging because it ensures that the collection always contains valid elements. Related methods include **checkedSet( )**, **checkedList( )**, **checkedMap( )**, and so on. They obtain a type-safe view for the indicated collection.

Notice that several methods, such as **synchronizedList( )** and **synchronizedSet( )**, are used to obtain synchronized (*thread-safe*) copies of the various collections. As a general rule, the standard collections implementations are not synchronized. You must use the synchronization algorithms to provide synchronization. One other point: iterators to synchronized collections must be used within **synchronized** blocks.

Method	Description
static <T> boolean addAll(Collection <? super T> *c*, T... *elements*)	Inserts the elements specified by *elements* into the collection specified by *c*. Returns **true** if the elements were added and **false** otherwise.
static <T> Queue<T> asLifoQueue(Deque<T> *c*)	Returns a last-in, first-out view of *c*.
static <T> int binarySearch(List<? extends T> *list*, T *value*, Comparator<? super T> *c*)	Searches for *value* in *list* ordered according to *c*. Returns the position of value in *list*, or a negative value if *value* is not found.
static <T> int binarySearch(List<? extends Comparable<? super T>> *list*, T *value*)	Searches for *value* in *list*. The list must be sorted. Returns the position of *value* in *list*, or a negative value if *value* is not found.
static <E> Collection<E> checkedCollection(Collection<E> *c*, Class<E> *t*)	Returns a run-time type-safe view of a collection. An attempt to insert an incompatible element will cause a **ClassCastException**.
static <E> List<E> checkedList(List<E> *c*, Class<E> *t*)	Returns a run-time type-safe view of a **List**. An attempt to insert an incompatible element will cause a **ClassCastException**.
static <K, V> Map<K, V> checkedMap(Map<K, V> *c*, Class<K> *keyT*, Class<V> *valueT*)	Returns a run-time type-safe view of a **Map**. An attempt to insert an incompatible element will cause a **ClassCastException**.
static <K, V> NavigableMap<K, V> checkedNavigableMap(NavigableMap<K, V> *nm*, Class<E> *keyT*, Class<V> *valueT*)	Returns a run-time type-safe view of a **NavigableMap**. An attempt to insert an incompatible element will cause a **ClassCastException**. (Added by JDK 8.)
static <E> NavigableSet<E> checkedNavigableSet(NavigableSet<E> *ns*, Class<E> *t*)	Returns a run-time type-safe view of a **NavigableSet**. An attempt to insert an incompatible element will cause a **ClassCastException**. (Added by JDK 8.)
static <E> Queue<E> checkedQueue(Queue<E> *q*, Class<E> *t*)	Returns a run-time type-safe view of a **Queue**. An attempt to insert an incompatible element will cause a **ClassCastException**. (Added by JDK 8.)
static <E> List<E> checkedSet(Set<E> *c*, Class<E> *t*)	Returns a run-time type-safe view of a **Set**. An attempt to insert an incompatible element will cause a **ClassCastException**.

Table 18-15 The Algorithms Defined by **Collections**

Method	Description
static <K, V> SortedMap<K, V> checkedSortedMap(SortedMap<K, V> *c*, Class<K> *keyT*, Class<V> *valueT*)	Returns a run-time type-safe view of a **SortedMap**. An attempt to insert an incompatible element will cause a **ClassCastException**.
static <E> SortedSet<E> checkedSortedSet(SortedSet<E> *c*, Class<E> *t*)	Returns a run-time type-safe view of a **SortedSet**. An attempt to insert an incompatible element will cause a **ClassCastException**.
static <T> void copy(List<? super T> *list1*, List<? extends T> *list2*)	Copies the elements of *list2* to *list1*.
static boolean disjoint(Collection<?> *a*, Collection<?> *b*)	Compares the elements in *a* to elements in *b*. Returns **true** if the two collections contain no common elements (i.e., the collections contain disjoint sets of elements). Otherwise, returns **false**.
static <T> Enumeration<T> emptyEnumeration( )	Returns an empty enumeration, which is an enumeration with no elements.
static <T> Iterator<T> emptyIterator( )	Returns an empty iterator, which is an iterator with no elements.
static <T> List<T> emptyList( )	Returns an immutable, empty **List** object of the inferred type.
static <T> ListIterator<T> emptyListIterator( )	Returns an empty list iterator, which is a list iterator that has no elements.
static <K, V> Map<K, V> emptyMap( )	Returns an immutable, empty **Map** object of the inferred type.
static <K, V> NavigableMap<K, V> emptyNavigableMap( )	Returns an immutable, empty **NavigableMap** object of the inferred type. (Added by JDK 8.)
static <E> NavigableSet<E> emptyNavigableSet( )	Returns an immutable, empty **NavigableSet** object of the inferred type. (Added by JDK 8.)
static <T> Set<T> emptySet( )	Returns an immutable, empty **Set** object of the inferred type.
static <K, V> SortedMap<K, V> emptySortedMap( )	Returns an immutable, empty **SortedMap** object of the inferred type. (Added by JDK 8.)
static <E> SortedSet<E> emptySortedSet( )	Returns an immutable, empty **SortedSet** object of the inferred type. (Added by JDK 8.)
static <T> Enumeration<T> enumeration(Collection<T> *c*)	Returns an enumeration over *c*. (See "The Enumeration Interface," later in this chapter.)
static <T> void fill(List<? super T> *list*, T *obj*)	Assigns *obj* to each element of *list*.
static int frequency(Collection<?> *c*, object *obj*)	Counts the number of occurrences of *obj* in *c* and returns the result.

Table 18-15 The Algorithms Defined by **Collections** *(continued)*

Method	Description
static int indexOfSubList(List<?> *list*, List<?> *subList*)	Searches *list* for the first occurrence of *subList*. Returns the index of the first match, or –1 if no match is found.
static int lastIndexOfSubList(List<?> *list*, List<?> *subList*)	Searches *list* for the last occurrence of *subList*. Returns the index of the last match, or –1 if no match is found.
static <T> ArrayList<T> list(Enumeration<T> *enum*)	Returns an **ArrayList** that contains the elements of *enum*.
static <T> T max(Collection<? extends T> *c*, Comparator<? super T> *comp*)	Returns the maximum element in *c* as determined by *comp*.
static <T extends Object & Comparable<? super T>> T max(Collection<? extends T> *c*)	Returns the maximum element in *c* as determined by natural ordering. The collection need not be sorted.
static <T> T min(Collection<? extends T> *c*, Comparator<? super T> *comp*)	Returns the minimum element in *c* as determined by *comp*. The collection need not be sorted.
static <T extends Object & Comparable<? super T>> T min(Collection<? extends T> *c*)	Returns the minimum element in *c* as determined by natural ordering.
static <T> List<T> nCopies(int *num*, T *obj*)	Returns *num* copies of *obj* contained in an immutable list. *num* must be greater than or equal to zero.
static <E> Set<E> newSetFromMap(Map<E, Boolean> *m*)	Creates and returns a set backed by the map specified by *m*, which must be empty at the time this method is called.
static <T> boolean replaceAll(List<T> *list*, T *old*, T *new*)	Replaces all occurrences of *old* with *new* in *list*. Returns **true** if at least one replacement occurred. Returns **false** otherwise.
static void reverse(List<T> *list*)	Reverses the sequence in *list*.
static <T> Comparator<T> reverseOrder(Comparator<T> *comp*)	Returns a reverse comparator based on the one passed in *comp*. That is, the returned comparator reverses the outcome of a comparison that uses *comp*.
static <T> Comparator<T> reverseOrder( )	Returns a reverse comparator, which is a comparator that reverses the outcome of a comparison between two elements.
static void rotate(List<T> *list*, int *n*)	Rotates *list* by *n* places to the right. To rotate left, use a negative value for *n*.
static void shuffle(List<T> *list*, Random *r*)	Shuffles (i.e., randomizes) the elements in *list* by using *r* as a source of random numbers.
static void shuffle(List<T> *list*)	Shuffles (i.e., randomizes) the elements in *list*.

Table 18-15 The Algorithms Defined by **Collections** *(continued)*

Method	Description
static <T> Set<T> singleton(T *obj*)	Returns *obj* as an immutable set. This is an easy way to convert a single object into a set.
static <T> List<T> singletonList(T *obj*)	Returns *obj* as an immutable list. This is an easy way to convert a single object into a list.
static <K, V> Map<K, V> singletonMap(K *k*, V *v*)	Returns the key/value pair *k/v* as an immutable map. This is an easy way to convert a single key/value pair into a map.
static <T> void sort(List<T> *list*, Comparator<? super T> *comp*)	Sorts the elements of *list* as determined by *comp*.
static <T extends Comparable<? super T>> void sort(List<T> *list*)	Sorts the elements of *list* as determined by their natural ordering.
static void swap(List<?> *list*, int *idx1*, int *idx2*)	Exchanges the elements in *list* at the indices specified by *idx1* and *idx2*.
static <T> Collection<T> synchronizedCollection(Collection<T> *c*)	Returns a thread-safe collection backed by *c*.
static <T> List<T> synchronizedList(List<T> *list*)	Returns a thread-safe list backed by *list*.
static <K, V> Map<K, V> synchronizedMap(Map<K, V> *m*)	Returns a thread-safe map backed by *m*.
static <K, V> NavigableMap<K, V> synchronizedNavigableMap(NavigableMap<K, V> *nm*)	Returns a synchronized navigable map backed by *nm*. (Added by JDK 8.)
static <T> NavigableSet<T> synchronizedNavigableSet(NavigableSet<T> *ns*)	Returns a synchronized navigable set backed by *ns*. (Added by JDK 8.)
static <T> Set<T> synchronizedSet(Set<T> *s*)	Returns a thread-safe set backed by *s*.
static <K, V> SortedMap<K, V> synchronizedSortedMap(SortedMap<K, V> *sm*)	Returns a thread-safe sorted map backed by *sm*.
static <T> SortedSet<T> synchronizedSortedSet(SortedSet<T> *ss*)	Returns a thread-safe sorted set backed by *ss*.
static <T> Collection<T> unmodifiableCollection(Collection<? extends T> *c*)	Returns an unmodifiable collection backed by *c*.
static <T> List<T> unmodifiableList(List<? extends T> *list*)	Returns an unmodifiable list backed by *list*.
static <K, V> Map<K, V> unmodifiableMap(Map<? extends K, ? extends V> *m*)	Returns an unmodifiable map backed by *m*.
static <K, V> NavigableMap<K, V> unmodifiableNavigableMap(NavigableMap<K, ? extends V> *nm*)	Returns an unmodifiable navigable map backed by *nm*. (Added by JDK 8.)

Table 18-15 The Algorithms Defined by **Collections** *(continued)*

Method	Description
static <T> NavigableSet<T> unmodifiableNavigableSet(NavigableSet<T> *ns*)	Returns an unmodifiable navigable set backed by *ns*. (Added by JDK 8.)
static <T> Set<T> unmodifiableSet(Set<? extends T> *s*)	Returns an unmodifiable set backed by *s*.
static <K, V> SortedMap<K, V> unmodifiableSortedMap(SortedMap<K, ? extends V> *sm*)	Returns an unmodifiable sorted map backed by *sm*.
static <T> SortedSet<T> unmodifiableSortedSet(SortedSet<T> *ss*)	Returns an unmodifiable sorted set backed by *ss*.

Table 18-15 The Algorithms Defined by **Collections** *(continued)*

The set of methods that begins with **unmodifiable** returns views of the various collections that cannot be modified. These will be useful when you want to grant some process read—but not write—capabilities on a collection.

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

The following program demonstrates some of the algorithms. It creates and initializes a linked list. The **reverseOrder( )** method returns a **Comparator** that reverses the comparison of **Integer** objects. The list elements are sorted according to this comparator and then are displayed. Next, the list is randomized by calling **shuffle( )**, and then its minimum and maximum values are displayed.

```
// Demonstrate various algorithms.
import java.util.*;

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

    // Create and initialize linked list.
    LinkedList<Integer> ll = new LinkedList<Integer>();
    ll.add(-8);
    ll.add(20);
    ll.add(-20);
    ll.add(8);

    // Create a reverse order comparator.
    Comparator<Integer> r = Collections.reverseOrder();

    // Sort list by using the comparator.
    Collections.sort(ll, r);

    System.out.print("List sorted in reverse: ");
    for(int i : ll)
      System.out.print(i+ " ");
```

```
    System.out.println();

    // Shuffle list.
    Collections.shuffle(ll);

    // Display randomized list.
    System.out.print("List shuffled: ");
    for(int i : ll)
      System.out.print(i + " ");

    System.out.println();
    System.out.println("Minimum: " + Collections.min(ll));
    System.out.println("Maximum: " + Collections.max(ll));
  }
}
```

Output from this program is shown here:

```
List sorted in reverse: 20 8 -8 -20
List shuffled: 20 -20 8 -8
Minimum: -20
Maximum: 20
```

Notice that **min( )** and **max( )** operate on the list after it has been shuffled. Neither requires a sorted list for its operation.

Arrays

The **Arrays** class provides various methods that are useful when working with arrays. These methods help bridge the gap between collections and arrays. 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 <T> List asList(T... *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. Here are some of its forms. (Additional forms let you search a subrange):

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 <T> int binarySearch(T[ ] *array*, T *value*, Comparator<? super T> *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 **copyOf( )** method returns a copy of an array and has the following forms:

```
static boolean[ ] copyOf(boolean[ ] source, int len)
static byte[ ] copyOf(byte[ ] source, int len)
static char[ ] copyOf(char[ ] source, int len)
static double[ ] copyOf(double[ ] source, int len)
static float[ ] copyOf(float[ ] source, int len)
static int[ ] copyOf(int[ ] source, int len)
static long[ ] copyOf(long[ ] source, int len)
static short[ ] copyOf(short[ ] source, int len)
static <T> T[ ] copyOf(T[ ] source, int len)
static <T,U> T[ ] copyOf(U[ ] source, int len, Class<? extends T[ ]> resultT)
```

The original array is specified by *source*, and the length of the copy is specified by *len*. If the copy is longer than *source*, then the copy is padded with zeros (for numeric arrays), **null**s (for object arrays), or **false** (for boolean arrays). If the copy is shorter than *source*, then the copy is truncated. In the last form, the type of *resultT* becomes the type of the array returned. If *len* is negative, a **NegativeArraySizeException** is thrown. If *source* is **null**, a **NullPointerException** is thrown. If *resultT* is incompatible with the type of *source*, an **ArrayStoreException** is thrown.

The **copyOfRange( )** method returns a copy of a range within an array and has the following forms:

```
static boolean[ ] copyOfRange(boolean[ ] source, int start, int end)
static byte[ ] copyOfRange(byte[ ] source, int start, int end)
static char[ ] copyOfRange(char[ ] source, int start, int end)
static double[ ] copyOfRange(double[ ] source, int start, int end)
static float[ ] copyOfRange(float[ ] source, int start, int end)
static int[ ] copyOfRange(int[ ] source, int start, int end)
static long[ ] copyOfRange(long[ ] source, int start, int end)
static short[ ] copyOfRange(short[ ] source, int start, int end)
static <T> T[ ] copyOfRange(T[ ] source, int start, int end)
static <T,U> T[ ] copyOfRange(U[ ] source, int start, int end,
                              Class<? extends T[ ]> resultT)
```

The original array is specified by *source*. The range to copy is specified by the indices passed via *start* and *end*. The range runs from *start* to *end* – 1. If the range is longer than *source*, then the copy is padded with zeros (for numeric arrays), **null**s (for object arrays), or **false** (for boolean arrays). In the last form, the type of *resultT* becomes the type of the array returned. If *start* is negative or greater than the length of *source*, an **ArrayIndexOutOfBoundsException** is thrown. If *start* is greater than *end*, an

IllegalArgumentException is thrown. If *source* is **null**, a **NullPointerException** is thrown. If *resultT* is incompatible with the type of *source*, an **ArrayStoreException** is thrown.

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 **deepEquals( )** method can be used to determine if two arrays, which might contain nested arrays, are equal. It has this declaration:

```
static boolean deepEquals(Object[ ] a, Object[ ] b)
```

It returns **true** if the arrays passed in *a* and *b* contain the same elements. If *a* and *b* contain nested arrays, then the contents of those nested arrays are also checked. It returns **false** if the arrays, or any nested arrays, differ.

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.

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 <T> void sort(T array[ ], Comparator<? super T> 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 last two forms can 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.

JDK 8 adds several new methods to **Arrays**. Perhaps the most important is **parallelSort( )** because it sorts, into ascending order, portions of an array in parallel and then merges the results. This approach can greatly speed up sorting times. Like **sort( )**, there are two basic types of **parallelSort( )**, each with several overloads. The first type sorts the entire array. It is shown here:

```
static void parallelSort(byte array[ ])
static void parallelSort(char array[ ])
static void parallelSort(double array[ ])
static void parallelSort(float array[ ])
static void parallelSort(int array[ ])
static void parallelSort(long array[ ])
static void parallelSort(short array[ ])
static <T extends Comparable<? super T>> void parallelSort(T array[ ])
static <T> void parallelSort(T array[ ], Comparator<? super T> c)
```

Here, *array* is the array to be sorted. In the last form, *c* is a comparator that is used to order the elements in the array. The last two forms can throw a **ClassCastException** if the elements of the array being sorted are not comparable. The second version of **parallelSort( )** enables you to specify a range within the array that you want to sort.

JDK 8 gives **Arrays** support for spliterators by including the **spliterator( )** method. It has two basic forms. The first type returns a spliterator to an entire array. It is shown here:

```
static Spliterator.OfDouble spliterator(double array[ ])
static Spliterator.OfInt spliterator(int array[ ])
static Spliterator.OfLong spliterator(long array[ ])
static <T> Spliterator spliterator(T array[ ])
```

Here, *array* is the array that the spliterator will cycle through. The second version of **spliterator( )** enables you to specify a range to iterate within the array.

Beginning with JDK 8, **Arrays** supports the new **Stream** interface (see Chapter 29) by including the **stream( )** method. It has two forms. The first is shown here:

```
static DoubleStream stream(double array[ ])
static IntStream stream(int array[ ])
static LongStream stream(long array[ ])
static <T> Stream stream(T array[ ])
```

Here, *array* is the array to which the stream will refer. The second version of **stream( )** enables you to specify a range within the array.

In addition to those just discussed, JDK 8 adds three other new methods. Two are related: **setAll( )** and **parallelSetAll( )**. Both assign values to all of the elements, but **parallelSetAll( )** works in parallel. Here is an example of each:

```
static void setAll(double array[ ],
                   IntToDoubleFunction<? extends T> genVal)

static void parallelSetAll(double array[ ],
                           IntToDoubleFunction<? extends T> genVal)
```

Several overloads exist for each of these that handle types **int**, **long**, and generic.

Finally, JDK 8 includes one of the more intriguing additions to **Arrays**. It is called **parallelPrefix( )**, and it modifies an array so that each element contains the cumulative result of an operation applied to all previous elements. For example, if the operation is multiplication, then on return, the array elements will contain the values associated with the running product of the original values. It has several overloads. Here is one example:

```
static void parallelPrefix(double array[ ], DoubleBinaryOperator func)
```

Here, *array* is the array being acted upon, and *func* specifies the operation applied. (**DoubleBinaryOperator** is a functional interface defined in **java.util.function**.) Many other versions are provided, including those that operate on types **int**, **long**, and generic, and those that let you specify a range within the array on which to operate.

Arrays also provides **toString( )** and **hashCode( )** for the various types of arrays. In addition, **deepToString( )** and **deepHashCode( )** are provided, which operate effectively on arrays that contain nested arrays.

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

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

    // Sort and display the array.
```

```
    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: array)
      System.out.print(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
```

The Legacy Classes and Interfaces

As explained at the start of this chapter, early versions of **java.util** did not include the Collections Framework. Instead, it defined several classes and an interface that provided an ad hoc method of storing objects. When collections were added (by J2SE 1.2), several of the original classes were reengineered to support the collection interfaces. Thus, they are now technically part of the Collections Framework. However, where a modern collection duplicates the functionality of a legacy class, you will usually want to use the newer collection class. In general, the legacy classes are supported because there is still code that uses them.

One other point: none of the modern collection classes described in this chapter are synchronized, but all the legacy classes are synchronized. This distinction may be important in some situations. Of course, you can easily synchronize collections by using one of the algorithms provided by **Collections**.

The legacy classes defined by **java.util** are shown here:

Dictionary	Hashtable	Properties	Stack	Vector

There is one legacy interface called **Enumeration**. The following sections examine **Enumeration** and each of the legacy classes, in turn.

The Enumeration Interface

The **Enumeration** interface defines the methods by which you can *enumerate* (obtain one at a time) the elements in a collection of objects. This legacy interface has been superseded by **Iterator**. Although not deprecated, **Enumeration** is considered obsolete for new code. However, it is used by several methods defined by the legacy classes (such as **Vector** and **Properties**) and is used by several other API classes. Because it is still in use, it was retrofitted for generics by JDK 5. It has this declaration:

```
interface Enumeration<E>
```

where **E** specifies the type of element being enumerated.

Enumeration specifies the following two methods:

```
boolean hasMoreElements( )
E nextElement( )
```

When implemented, **hasMoreElements( )** must return **true** while there are still more elements to extract, and **false** when all the elements have been enumerated. **nextElement( )** returns the next object in the enumeration. That is, each call to **nextElement( )** obtains the next object in the enumeration. It throws **NoSuchElementException** when the enumeration is complete.

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 duplicate the functionality of methods defined by the Collections Framework. With the advent of collections, **Vector** was reengineered to extend **AbstractList** and to implement the **List** interface. With the release of JDK 5, it was retrofitted for generics and reengineered to implement **Iterable**. This means that **Vector** is fully compatible with collections, and a **Vector** can have its contents iterated by the enhanced **for** loop.

Vector is declared like this:

```
class Vector<E>
```

Here, **E** specifies the type of element that will be stored.

Here are the **Vector** constructors:

```
Vector( )
Vector(int size)
Vector(int size, int incr)
Vector(Collection<? extends E> c)
```

The first form creates a default vector, which has an initial size of 10. The second form creates a vector whose initial capacity is specified by *size*. The third form 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. The fourth form creates a vector that contains the elements of collection *c*.

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 as the vector grows. 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**.

In addition to the collections methods specified by **List**, **Vector** defines several legacy methods, which are summarized in Table 18-16.

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( )**.

Method	Description
void addElement(E *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 returns **false** if it is not.
void copyInto(Object *array*[ ])	The elements contained in the invoking vector are copied into the array specified by *array*.
E elementAt(int *index*)	Returns the element at the location specified by *index*.
Enumeration<E> elements( )	Returns an enumeration of the elements in the vector.
void ensureCapacity(int *size*)	Sets the minimum capacity of the vector to *size*.
E 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.

Table 18-16 The Legacy Methods Defined by **Vector**

Method	Description
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(E *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.
E lastElement( )	Returns the last element in the vector.
int lastIndexOf(Object *element*)	Returns the 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(E *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( )	Returns the number of elements currently in the vector.
String toString( )	Returns the string equivalent of the vector.
void trimToSize( )	Sets the vector's capacity equal to the number of elements that it currently holds.

Table 18-16 The Legacy Methods Defined by **Vector** *(continued)*

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<Integer> v = new Vector<Integer>(3, 2);

    System.out.println("Initial size: " + v.size());
    System.out.println("Initial capacity: " +
                       v.capacity());

    v.addElement(1);
    v.addElement(2);
    v.addElement(3);
    v.addElement(4);

    System.out.println("Capacity after four additions: " +
                       v.capacity());

    v.addElement(5);
    System.out.println("Current capacity: " +
                       v.capacity());

    v.addElement(6);
    v.addElement(7);

    System.out.println("Current capacity: " +
                       v.capacity());

    v.addElement(9);
    v.addElement(10);

    System.out.println("Current capacity: " +
                       v.capacity());

    v.addElement(11);
    v.addElement(12);

    System.out.println("First element: " + v.firstElement());
    System.out.println("Last element: " + v.lastElement());

    if(v.contains(3))
      System.out.println("Vector contains 3.");

    // Enumerate the elements in the vector.
    Enumeration<Integer> vEnum = v.elements();

    System.out.println("\nElements in vector:");
    while(vEnum.hasMoreElements())
      System.out.print(vEnum.nextElement() + " ");
    System.out.println();
  }
}
```