



RV Institute of Technology and Management®

RV Educational Institutions®

RV Institute of Technology and Management

(Affiliated to VTU, Belagavi)

JP Nagar 8th Phase, Bengaluru - 560076

Department of Information Science and Engineering



Course Name: Advanced Java

Course Code: BIS402

Module 1 – The collections and Framework

IV Semester

2022 Scheme

Prepared By :

Dr. Vinod Kumar M & Dr. Kirankumar K,

Associate & Assistant Professor,

Department of Information

Science and Engineering RVITM, Bengaluru -560076



Collections Overview

The Java Collections Framework standardizes the way in which groups of objects are handled by your programs. Collections were not part of the original Java release, but were added by J2SE 1.2. Prior to the Collections Framework, Java provided ad hoc classes such as **Dictionary**, **Vector**, **Stack**, and **Properties** to store and manipulate groups of objects. Although these classes were quite useful, they lacked a central, unifying theme. The way that you used **Vector** was different from the way that you used **Properties**, for example. Also, this early, ad hoc approach was not designed to be easily extended or adapted. Collections were an answer to these (and other) problems.

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

Algorithms are another important part of the collection mechanism. Algorithms operate on collections and are defined as static methods within the **Collections** class. Thus, they are available for all collections. Each collection class need not implement its own versions. The algorithms provide a standard means of manipulating collections.



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

JDK 8 added another type of iterator called a *spliterator*. In brief, spliterators are iterators that provide support for parallel iteration. The interfaces that support spliterators are **Spliterator** and several nested interfaces that support primitive types. JDK 8 also added iterator interfaces designed for use with primitive types, such as **PrimitiveIterator** and **PrimitiveIterator.OfDouble**.

In addition to collections, the framework defines several map interfaces and classes. *Maps* store key/value pairs. Although maps are part of the Collections Framework, they are not “collections” in the strict use of the term. You can, however, obtain a *collection-view* of a map. Such a view contains the elements from the map stored in a collection. Thus, you can process the contents of a map as a collection, if you choose.

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

The Collection Interfaces

The Collections Framework defines several core interfaces. This section provides an overview of each interface. Beginning with the collection interfaces is necessary because they determine the fundamental nature of



the collection classes. Put differently, the concrete classes simply provide different implementations of the standard interfaces. The interfaces that underpin collections are summarized in the following table:

Interface	Description
Collection	Enables you to work with groups of objects; it is at the top of the collections hierarchy.
Deque	Extends Queue to handle a double-ended queue.
List	Extends Collection to handle sequences (lists of objects).
NavigableSet	Extends SortedSet to handle retrieval of elements based on closest-match searches.
Queue	Extends Collection to handle special types of lists in which elements are removed only from the head.
Set	Extends Collection to handle sets, which must contain unique elements.
SortedSet	Extends Set to handle sorted sets.

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

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

The Collection Interface

The **Collection** interface is the foundation upon which the Collections Framework is built because it must be implemented by any class that defines a collection. **Collection** is a generic interface that has this declaration:



interface Collection<E>

Here, **E** specifies the type of objects that the collection will hold. **Collection** extends the **Iterable** interface. This means that all collections can be cycled through by use of the for-each style **for** loop. (Recall that only classes that implement **Iterable** can be cycled through by the **for**.)

Collection declares the core methods that all collections will have. These methods are summarized in Table 1-1. Because all collections implement **Collection**, familiarity with its methods is necessary for a clear understanding of the framework. Several of these methods can throw an **UnsupportedOperationException**. As explained, this occurs if a collection cannot be modified. A **ClassCastException** is generated when one object is incompatible with another, such as when an attempt is made to add an incompatible object to a collection. A **NullPointerException** is thrown if an attempt is made to store a **null** object and **null** elements are not allowed in the collection. An **IllegalArgumentException** is thrown if an invalid argument is used. An **IllegalStateException** is thrown if an attempt is made to add an element to a fixed-length collection that is full.

**Table 1-1 The Methods Declared by Collection**

Method	Description
boolean add(E <i>obj</i>)	Adds <i>obj</i> to the invoking collection. Returns true if <i>obj</i> was added to the collection. Returns false if <i>obj</i> is already a member of the collection and the collection does not allow duplicates.
boolean addAll(Collection<? extends E> <i>c</i>)	Adds all the elements of <i>c</i> to the invoking collection. Returns true if the collection changed (i.e., the elements were added). Otherwise, returns false .
void clear()	Removes all elements from the invoking collection.
boolean contains(Object <i>obj</i>)	Returns true if <i>obj</i> is an element of the invoking collection. Otherwise, returns false .
boolean containsAll(Collection<?> <i>c</i>)	Returns true if the invoking collection contains all elements of <i>c</i> . Otherwise, returns false .
boolean equals(Object <i>obj</i>)	Returns true if the invoking collection and <i>obj</i> are equal. Otherwise, returns false .
int hashCode()	Returns the hash code for the invoking collection.
boolean isEmpty()	Returns true if the invoking collection is empty. Otherwise, returns false .
Iterator<E> iterator()	Returns an iterator for the invoking collection.
default Stream<E> parallelStream()	Returns a stream that uses the invoking collection as its source for elements. If possible, the stream supports parallel operations.
boolean remove(Object <i>obj</i>)	Removes one instance of <i>obj</i> from the invoking collection. Returns true if the element was removed. Otherwise, returns false .
boolean removeAll(Collection<?> <i>c</i>)	Removes all elements of <i>c</i> from the invoking collection. Returns true if the collection changed (i.e., elements were removed). Otherwise, returns false .
default boolean removeIf(Predicate<? super E> <i>predicate</i>)	Removes from the invoking collection those elements that satisfy the condition specified by <i>predicate</i> .
boolean retainAll(Collection<?> <i>c</i>)	Removes all elements from the invoking collection except those in <i>c</i> . Returns true if the collection changed (i.e., elements were removed). Otherwise, returns false .
int size()	Returns the number of elements held in the invoking collection.
default Spliterator<E> spliterator()	Returns a spliterator to the invoking collections.



default Stream<E> stream()	Returns a stream that uses the invoking collection as its source for elements. The stream is sequential.
default <T> T[] toArray(IntFunction<T[]> arrayGen)	Returns an array of the elements from the invoking collection. The returned array is created by the function specified by <i>arrayGen</i> . An ArrayStoreException is thrown if any collection element has a type that is not compatible with the array type. (Added by JDK 11.)
Object[] toArray()	Returns an array of the elements from the invoking collection.
<T> T[] toArray(T array[])	Returns an array of the elements from the invoking collection. If the size of <i>array</i> equals the number of elements, these are returned in <i>array</i> . If the size of <i>array</i> is less than the number of elements, a new array of the necessary size is allocated and returned. If the size of <i>array</i> is greater than the number of elements, the array element following the last collection element is set to null . An ArrayStoreException is thrown if any collection element has a type that is not compatible with the array type.

Objects are added to a collection by calling **add()**. Notice that **add()** takes an argument of type **E**, which means that objects added to a collection must be compatible with the type of data expected by the collection. You can add the entire contents of one collection to another by calling **addAll()**.

You can remove an object by using **remove()**. To remove a group of objects, call **removeAll()**. You can remove all elements except those of a specified group by calling **retainAll()**. To remove an element only if it satisfies some condition, you can use **removeIf()**. To empty a collection, call **clear()**.

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

The **toArray()** methods return an array that contains the elements stored in the collection. The first returns an array of **Object**. The second returns an array of elements that have the same type as the array specified as a



parameter.

Normally, the second form is more convenient because it returns the desired array type. Beginning with JDK 11, a third form has been added that lets you specify a function that obtains the array. These methods are more important than it might at first seem. Often, processing the contents of a collection by using array-like syntax is advantageous. By providing a pathway between collections and arrays, you can have the best of both worlds.

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

Another important method is **iterator()**, which returns an iterator to a collection. The **spliterator()** method returns a spliterator to the collection. Iterators are frequently used when working with collections. Finally, the **stream()** and **parallelStream()** methods return a **Stream** that uses the collection as a source of elements.

The List Interface

The **List** interface extends **Collection** and declares the behavior of a collection that stores a sequence of elements. Elements can be inserted or accessed by their position in the list, using a zero-based index. A list may contain duplicate elements. **List** is a generic interface that has this declaration:

```
interface List<E>
```

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

In addition to the methods defined by **Collection**, **List** defines some of its own, which are summarized in Table 1-2. Note again that several of these methods will throw an **UnsupportedOperationException** if the list cannot be modified, and a **ClassCastException** is generated when one object is incompatible with another, such as when an attempt is made to add an incompatible object to a list. Also, several methods will throw an



IndexOutOfBoundsException if an invalid index is used. A **NullPointerException** is thrown if an attempt is made to store a **null** object and **null** elements are not allowed in the list. An **IllegalArgumentException** is thrown if an invalid argument is used.

**Table 1-2 The Methods Declared by List**

Method	Description
void add(int <i>index</i> , E <i>obj</i>)	Inserts <i>obj</i> into the invoking list at the index passed in <i>index</i> . Any preexisting elements at or beyond the point of insertion are shifted up. Thus, no elements are overwritten.
boolean addAll(int <i>index</i> , Collection<? extends E> <i>c</i>)	Inserts all elements of <i>c</i> into the invoking list at the index passed in <i>index</i> . Any preexisting elements at or beyond the point of insertion are shifted up. Thus, no elements are overwritten. Returns true if the invoking list changes and returns false otherwise.
static <E> List<E> copyOf(Collection<? extends E> <i>from</i>)	Returns a list that contains the same elements as that specified by <i>from</i> . The returned list is unmodifiable. Null values are not allowed.
E get(int <i>index</i>)	Returns the object stored at the specified index within the invoking collection.
int indexOf(Object <i>obj</i>)	Returns the index of the first instance of <i>obj</i> in the invoking list. If <i>obj</i> is not an element of the list, -1 is returned.
int lastIndexOf(Object <i>obj</i>)	Returns the index of the last instance of <i>obj</i> in the invoking list. If <i>obj</i> is not an element of the list, -1 is returned.
ListIterator<E> listIterator()	Returns an iterator to the start of the invoking list.
ListIterator<E> listIterator(int <i>index</i>)	Returns an iterator to the invoking list that begins at the specified <i>index</i> .
static <E> List<E> of(parameter-list)	Creates an unmodifiable list containing the elements specified in <i>parameter-list</i> . Null elements are not allowed. Many overloaded versions are provided. See the discussion in the text for details.
E remove(int <i>index</i>)	Removes the element at position <i>index</i> from the invoking list and returns the deleted element. The resulting list is compacted. That is, the indexes of subsequent elements are decremented by one.
default void replaceAll(UnaryOperator<E> <i>opToApply</i>)	Updates each element in the list with the value obtained from the <i>opToApply</i> function.
E set(int <i>index</i> , E <i>obj</i>)	Assigns <i>obj</i> to the location specified by <i>index</i> within the invoking list. Returns the old value.
default void sort(Comparator<? super E> <i>comp</i>)	Sorts the list using the comparator specified by <i>comp</i> .
List<E> subList(int <i>start</i> , int <i>end</i>)	Returns a list that includes elements from <i>start</i> to <i>end</i> -1 in the invoking list. Elements in the returned list are also referenced by the invoking object.



To the versions of **add()** and **addAll()** defined by **Collection**, **List** adds the methods **add(int, E)** and **addAll(int, Collection)**. These methods insert elements at the specified index. Also, the semantics of **add(E)** and **addAll(Collection)** defined by **Collection** are changed by **List** so that they add elements to the end of the list. You can modify each element in the collection by using **replaceAll()**.

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

You can obtain a sublist of a list by calling **subList()**, specifying the beginning and ending indexes of the sublist. As you can imagine, **subList()** makes list processing quite convenient. One way to sort a list is with the **sort()** method defined by **List**.

Beginning with JDK 9, **List** includes the **of()** factory method, which has a number of overloads. Each version returns an unmodifiable, value-based collection that is comprised of the arguments that it is passed. The primary purpose of **of()** is to provide a convenient, efficient way to create a small **List** collection. There are 12 overloads of **of()**. One takes no arguments and creates an empty list. It is shown here:

```
static <E> List<E> of( )
```

Ten overloads take from 1 to 10 arguments and create a list that contains the specified elements. They are shown here:

```
static <E> List<E> of(E obj1)
```

```
static <E> List<E> of(E obj1, E obj2)
```

```
static <E> List<E> of(E obj, E obj2, E obj3)
```

```
...
```

```
static <E> List<E> of(E obj1, E obj2, E obj3, E obj4, E obj5,  
E obj6, E obj7, E obj8, E obj9, E obj10)
```



The final **of()** overload specifies a varargs parameter that takes an arbitrary number of elements or an array of elements. It is shown here:

```
static <E> List<E> of(E ... objs)
```

For all versions, **null** elements are not allowed. In all cases, the **List** implementation is unspecified.

The Set Interface

The **Set** interface defines a set. It extends **Collection** and specifies the behavior of a collection that does not allow duplicate elements. Therefore, the **add()** method returns **false** if an attempt is made to add duplicate elements to a set.

With two exceptions, it does not specify any additional methods of its own. **Set** is a generic interface that has this declaration:

```
interface Set<E>
```

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

Beginning with JDK 9, **Set** includes the **of()** factory method, which has a number of overloads. Each version returns an unmodifiable, value-based collection that is comprised of the arguments that it is passed. The primary purpose of **of()** is to provide a convenient, efficient way to create a small **Set** collection. There are 12 overloads of **of()**. One takes no arguments and creates an empty set. It is shown here:

```
static <E> Set<E> of()
```

Ten overloads take from 1 to 10 arguments and create a list that contains the specified elements. They are shown here:



```
static <E> Set<E> of(E obj1)
static <E> Set<E> of(E obj1, E obj2)
static <E> Set<E> of(E obj, E obj2, E obj3)
...
static <E> Set<E> of(E obj1, E obj2, E obj3, E obj4, E obj5,
                      E obj6, E obj7, E obj8, E obj9, E obj10)
```

The final **of()** overload specifies a varargs parameter that takes an arbitrary number of elements or an array of elements. It is shown here:

```
static <E> Set<E> of(E ... objs)
```

For all versions, **null** elements are not allowed. In all cases, the **Set** implementation is unspecified.

Beginning with JDK 10, **Set** includes the static **copyOf()** method shown here:

```
static <E> Set<E> copyOf(Collection <? extends E> from)
```

It returns a set that contains the same elements as *from*. Null values are not allowed. The returned set is unmodifiable.

The SortedSet Interface

The **SortedSet** interface extends **Set** and declares the behavior of a set sorted in ascending order. **SortedSet** is a generic interface that has this declaration:

```
interface SortedSet<E>
```

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

In addition to those methods provided by **Set**, the **SortedSet** interface declares the methods summarized in Table 1-3. Several methods throw a **NoSuchElementException** when no items are contained in the invoking set. A **ClassCastException** is thrown when an object is incompatible with the elements in a set. A **NullPointerException** is thrown if an attempt is made to use a **null** object and **null** is not allowed in the set. An



IllegalArgumentException is thrown if an invalid argument is used.

Table 1-3 The Methods Declared by **SortedSet**

Method	Description
Comparator<? super E> comparator()	Returns the invoking sorted set's comparator. If the natural ordering is used for this set, null is returned.
E first()	Returns the first element in the invoking sorted set.
SortedSet<E> headSet(E end)	Returns a SortedSet containing those elements less than <i>end</i> that are contained in the invoking sorted set. Elements in the returned sorted set are also referenced by the invoking sorted set.
E last()	Returns the last element in the invoking sorted set.
SortedSet<E> subSet(E start, E end)	Returns a SortedSet that includes those elements between <i>start</i> and <i>end</i> -1. Elements in the returned collection are also referenced by the invoking object.
SortedSet<E> tailSet(E start)	Returns a SortedSet that contains those elements greater than or equal to <i>start</i> that are contained in the sorted set. Elements in the returned set are also referenced by the invoking object.

SortedSet defines several methods that make set processing more convenient. To obtain the first object in the set, call **first()**. To get the last element, use **last()**. You can obtain a subset of a sorted set by calling **subSet()**, specifying the first and last object in the set. If you need the subset that starts with the first element in the set, use **headSet()**. If you want the subset that ends the set, use **tailSet()**.

The NavigableSet Interface

The **NavigableSet** interface extends **SortedSet** and declares the behavior of a collection that supports the retrieval of elements based on the closest match to a given value or values. **NavigableSet** is a generic interface that has this declaration:

```
interface NavigableSet<E>
```

Here, **E** specifies the type of objects that the set will hold. In addition to the methods that it inherits from **SortedSet**, **NavigableSet** adds those summarized



in Table 1-4. A **ClassCastException** is thrown when an object is incompatible with the elements in the set. A **NullPointerException** is thrown if an attempt is made to use a **null** object and **null** is not allowed in the set. An **IllegalArgumentException** is thrown if an invalid argument is used.

Table 1-4 The Methods Declared by **NavigableSet**

Method	Description
E ceiling(E <i>obj</i>)	Searches the set for the smallest element <i>e</i> such that <i>e</i> >= <i>obj</i> . If such an element is found, it is returned. Otherwise, null is returned.
Iterator<E> descendingIterator()	Returns an iterator that moves from the greatest to least. In other words, it returns a reverse iterator.
NavigableSet<E> descendingSet()	Returns a NavigableSet that is the reverse of the invoking set. The resulting set is backed by the invoking set.
E floor(E <i>obj</i>)	Searches the set for the largest element <i>e</i> such that <i>e</i> <= <i>obj</i> . If such an element is found, it is returned. Otherwise, null is returned.
NavigableSet<E> headSet(E <i>upperBound</i> , boolean <i>incl</i>)	Returns a NavigableSet that includes all elements from the invoking set that are less than <i>upperBound</i> . If <i>incl</i> is true , then an element equal to <i>upperBound</i> is included. The resulting set is backed by the invoking set.
E higher(E <i>obj</i>)	Searches the set for the smallest element <i>e</i> such that <i>e</i> > <i>obj</i> . If such an element is found, it is returned. Otherwise, null is returned.
E lower(E <i>obj</i>)	Searches the set for the largest element <i>e</i> such that <i>e</i> < <i>obj</i> . If such an element is found, it is returned. Otherwise, null is returned.
E pollFirst()	Returns the first element, removing the element in the process. Because the set is sorted, this is the element with the least value. null is returned if the set is empty.
E pollLast()	Returns the last element, removing the element in the process. Because the set is sorted, this is the element with the greatest value. null is returned if the set is empty.
NavigableSet<E> subSet(E <i>lowerBound</i> , boolean <i>lowIncl</i> , E <i>upperBound</i> , boolean <i>highIncl</i>)	Returns a NavigableSet that includes all elements from the invoking set that are greater than <i>lowerBound</i> and less than <i>upperBound</i> . If <i>lowIncl</i> is true , then an element equal to <i>lowerBound</i> is included. If <i>highIncl</i> is true , then an element equal to <i>upperBound</i> is included. The resulting set is backed by the invoking set.
NavigableSet<E> tailSet(E <i>lowerBound</i> , boolean <i>incl</i>)	Returns a NavigableSet that includes all elements from the invoking set that are greater than <i>lowerBound</i> . If <i>incl</i> is true , then an element equal to <i>lowerBound</i> is included. The resulting set is backed by the invoking set.



The Queue Interface

The **Queue** interface extends **Collection** and declares the behavior of a queue, which is often a first-in, first-out list. However, there are types of queues in which the ordering is based upon other criteria. **Queue** is a generic interface that has this declaration:

```
interface Queue<E>
```

Here, **E** specifies the type of objects that the queue will hold. The methods declared by **Queue** are shown in Table 1-5.

Table 1-5 The Methods Declared by Queue

Method	Description
E element()	Returns the element at the head of the queue. The element is not removed. It throws NoSuchElementException if the queue is empty.
boolean offer(E obj)	Attempts to add <i>obj</i> to the queue. Returns true if <i>obj</i> was added and false otherwise.
E peek()	Returns the element at the head of the queue. It returns null if the queue is empty. The element is not removed.
E poll()	Returns the element at the head of the queue, removing the element in the process. It returns null if the queue is empty.
E remove()	Removes the element at the head of the queue, returning the element in the process. It throws NoSuchElementException if the queue is empty.

Several methods throw a **ClassCastException** when an object is incompatible with the elements in the queue. A **NullPointerException** is thrown if an attempt is made to store a **null** object and **null** elements are not allowed in the queue. An **IllegalArgumentException** is thrown if an invalid argument is used. An **IllegalStateException** is thrown if an attempt is made to add an element to a fixed-length queue that is full. A **NoSuchElementException** is thrown if an attempt is made to remove an element from an empty queue.

Despite its simplicity, **Queue** offers several points of interest. First, elements can only be removed from the head of the queue. Second, there



are two methods that obtain and remove elements: **poll()** and **remove()**. The difference between them is that **poll()** returns **null** if the queue is empty, but **remove()** throws an exception. Third, there are two methods, **element()** and **peek()**, that obtain but don't remove the element at the head of the queue. They differ only in that **element()** throws an exception if the queue is empty, but **peek()** returns **null**. Finally, notice that **offer()** only attempts to add an element to a queue. Because some queues have a fixed length and might be full, **offer()** can fail.

The Deque Interface

The **Deque** interface extends **Queue** and declares the behavior of a double-ended queue. Double-ended queues can function as standard, first-in, first-out queues or as last-in, first-out stacks. **Deque** is a generic interface that has this declaration:

```
interface Deque<E>
```

Here, **E** specifies the type of objects that the deque will hold. In addition to the methods that it inherits from **Queue**, **Deque** adds those methods summarized in Table 1-6. Several methods throw a **ClassCastException** when an object is incompatible with the elements in the deque. A **NullPointerException** is thrown if an attempt is made to store a **null** object and **null** elements are not allowed in the deque. An **IllegalArgumentException** is thrown if an invalid argument is used. An **IllegalStateException** is thrown if an attempt is made to add an element to a fixed-length deque that is full. A **NoSuchElementException** is thrown if an attempt is made to remove an element from an empty deque.

**Table 1-6 The Methods Declared by Deque**

Method	Description
void addFirst(E <i>obj</i>)	Adds <i>obj</i> to the head of the deque. Throws an IllegalStateException if a capacity-restricted deque is out of space.
void addLast(E <i>obj</i>)	Adds <i>obj</i> to the tail of the deque. Throws an IllegalStateException if a capacity-restricted deque is out of space.
Iterator<E> descendingIterator()	Returns an iterator that moves from the tail to the head of the deque. In other words, it returns a reverse iterator.
E getFirst()	Returns the first element in the deque. The object is not removed from the deque. It throws NoSuchElementException if the deque is empty.
E getLast()	Returns the last element in the deque. The object is not removed from the deque. It throws NoSuchElementException if the deque is empty.
boolean offerFirst(E <i>obj</i>)	Attempts to add <i>obj</i> to the head of the deque. Returns true if <i>obj</i> was added and false otherwise. Therefore, this method returns false when an attempt is made to add <i>obj</i> to a full, capacity-restricted deque.
boolean offerLast(E <i>obj</i>)	Attempts to add <i>obj</i> to the tail of the deque. Returns true if <i>obj</i> was added and false otherwise.
E peekFirst()	Returns the element at the head of the deque. It returns null if the deque is empty. The object is not removed.
E peekLast()	Returns the element at the tail of the deque. It returns null if the deque is empty. The object is not removed.
E pollFirst()	Returns the element at the head of the deque, removing the element in the process. It returns null if the deque is empty.
E pollLast()	Returns the element at the tail of the deque, removing the element in the process. It returns null if the deque is empty.
E pop()	Returns the element at the head of the deque, removing it in the process. It throws NoSuchElementException if the deque is empty.

Notice that **Deque** includes the methods **push()** and **pop()**. These methods enable a **Deque** to function as a stack. Also, notice the **descendingIterator()** method. It returns an iterator that returns elements in reverse order. In



other words, it returns an iterator that moves from the end of the collection to the start. A **Deque** implementation can be *capacity-restricted*, which means that only a limited number of elements can be added to the deque. When this is the case, an attempt to add an element to the deque can fail. **Deque** allows you to handle such a failure in two ways. First, methods such as **addFirst()** and **addLast()** throw an **IllegalStateException** if a capacity-restricted deque is full. Second, methods such as **offerFirst()** and **offerLast()** return **false** if the element cannot be added.

The Collection Classes

Now that you are familiar with the collection interfaces, you are ready to examine the standard classes that implement them. Some of the classes provide full implementations that can be used as-is. Others are abstract, providing skeletal implementations that are used as starting points for creating concrete collections. As a general rule, the collection classes are not synchronized, but as you will see later in this chapter, it is possible to obtain synchronized versions.

The core collection classes are summarized in the following table:



Class	Description
AbstractCollection	Implements most of the Collection interface.
AbstractList	Extends AbstractCollection and implements most of the List interface.
AbstractQueue	Extends AbstractCollection and implements parts of the Queue interface.
AbstractSequentialList	Extends AbstractList for use by a collection that uses sequential rather than random access of its elements.
LinkedList	Implements a linked list by extending AbstractSequentialList .
ArrayList	Implements a dynamic array by extending AbstractList .
ArrayDeque	Implements a dynamic double-ended queue by extending AbstractCollection and implementing the Deque interface.
AbstractSet	Extends AbstractCollection and implements most of the Set interface.
EnumSet	Extends AbstractSet for use with enum elements.
HashSet	Extends AbstractSet for use with a hash table.
LinkedHashSet	Extends HashSet to allow insertion-order iterations.
PriorityQueue	Extends AbstractQueue to support a priority-based queue.
TreeSet	Implements a set stored in a tree. Extends AbstractSet .

The ArrayList Class

The **ArrayList** class extends **AbstractList** and implements the **List** interface.

ArrayList is a generic class that has this declaration:

```
class ArrayList<E>
```

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

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



ArrayList has the constructors shown here:

```
ArrayList( )
ArrayList(Collection<? extends E> c) ArrayList(int
capacity)
```

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

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



```
// Demonstrate ArrayList.  
import java.util.*;  
  
class ArrayListDemo {  
    public static void main(String args[]) {  
        // Create an array list.  
        ArrayList<String> al = new ArrayList<String>();  
  
        System.out.println("Initial size of al: " +  
                           al.size());  
  
        // Add elements to the array list.  
        al.add("C");  
        al.add("A");  
        al.add("E");  
        al.add("B");  
        al.add("D");  
        al.add("F");  
        al.add(1, "A2");  
  
        System.out.println("Size of al after additions: " +  
                           al.size());  
  
        // Display the array list.  
        System.out.println("Contents of al: " + al);  
  
        // Remove elements from the array list.  
        al.remove("F");  
        al.remove(2);  
  
        System.out.println("Size of al after deletions: " +  
                           al.size());  
  
        System.out.println("Contents of al: " + al);  
    }  
}
```



The output from this program is shown here:

Initial size of al: 0

Size of al after additions: 7

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

Contents of al: [C, A2, E, B, D]

Notice that **a1** starts out empty and grows as elements are added to it. When elements are removed, its size is reduced.

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

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

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

Here, *cap* specifies the new minimum capacity of the collection.

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



```
void trimToSize( )
```

Obtaining an Array from an ArrayList

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

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

Whatever the reason, converting an **ArrayList** to an array is a trivial matter.

As explained earlier, there are three versions of **toArray()**, which are shown again here for your convenience:

```
object[ ] toArray( )
<T> T[ ] toArray(T array[ ])
default <T> T[ ] toArray(IntFunction<T[ ]> arrayGen)
```

The first returns an array of **Object**. The second and third forms return an array of elements that have the same type as **T**. Here, we will use the second form because of its convenience. The following program shows it in action.



```
// Convert an ArrayList into an array.  
import java.util.*;  
  
class ArrayListToArray {  
    public static void main(String args[]) {  
        // Create an array list.  
        ArrayList<Integer> al = new ArrayList<Integer>();  
  
        // Add elements to the array list.  
        al.add(1);  
        al.add(2);  
        al.add(3);  
        al.add(4);  
  
        System.out.println("Contents of al: " + al);  
  
        // Get the array.  
        Integer ia[] = new Integer[al.size()];  
        ia = al.toArray(ia);  
  
        int sum = 0;  
  
        // Sum the array.  
        for(int i : ia) sum += i;  
  
        System.out.println("Sum is: " + sum);  
    }  
}
```

The output from the program is shown here:

Contents of al: [1, 2, 3, 4]

Sum is: 10

The program begins by creating a collection of integers. Next, **toArray()** is called and it obtains an array of **Integers**. Then, the contents of that array are summed by use of a for-each style **for** loop.

There is something else of interest in this program. As you know, collections



can store only references, not values of primitive types. However, autoboxing makes it possible to pass values of type **int** to **add()** without having to manually wrap them within an **Integer**, as the program shows. Autoboxing causes them to be automatically wrapped. In this way, autoboxing significantly improves the ease with which collections can be used to store primitive values.

The **LinkedList** Class

The **LinkedList** class extends **AbstractSequentialList** and implements the **List**, **Deque**, and **Queue** interfaces. It provides a linked-list data structure. **LinkedList** is a generic class that has this declaration:

```
class LinkedList<E>
```

Here, **E** specifies the type of objects that the list will hold. **LinkedList** has the two constructors shown here:

```
LinkedList( )
LinkedList(Collection<? extends E> c)
```

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

Because **LinkedList** implements the **Deque** interface, you have access to the methods defined by **Deque**. For example, to add elements to the start of a list, you can use **addFirst()** or **offerFirst()**. To add elements to the end of the list, use **addLast()** or **offerLast()**. To obtain the first element, you can use **getFirst()** or **peekFirst()**. To obtain the last element, use **getLast()** or **peekLast()**. To remove the first element, use **removeFirst()** or **pollFirst()**. To remove the last element, use **removeLast()** or **pollLast()**.

The following program illustrates **LinkedList**:



```
// Demonstrate LinkedList.  
import java.util.*;  
  
class LinkedListDemo {  
    public static void main(String args[]) {  
        // Create a linked list.  
        LinkedList<String> ll = new LinkedList<String>();  
  
        // Add elements to the linked list.  
        ll.add("F");  
        ll.add("B");  
        ll.add("D");  
        ll.add("E");  
        ll.add("C");  
        ll.addLast("Z");  
        ll.addFirst("A");  
  
        ll.add(1, "A2");  
  
        System.out.println("Original contents of ll: " + ll);  
  
        // Remove elements from the linked list.  
        ll.remove("F");  
        ll.remove(2);  
  
        System.out.println("Contents of ll after deletion: "  
                           + ll);  
    }  
}
```



```
// Remove first and last elements.  
ll.removeFirst();  
ll.removeLast();  
  
System.out.println("ll after deleting first and last: "  
+ ll);  
  
// Get and set a value.  
  
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 factor*) 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. **PriorityQueues** are dynamic, growing as necessary.

PriorityQueue defines the seven constructors shown here:

```
PriorityQueue()
PriorityQueue(int capacity)
PriorityQueue(Comparator<? super E> comp) 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 1-7 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.

Table 1-7 The Methods Declared by **EnumSet**

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



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 1-8. The methods declared by **ListIterator** (along with those inherited from **Iterator**) are shown in Table 1-9. 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.

Table 1-8 The Methods Declared by **Iterator**

Method	Description
default void forEachRemaining(Consumer<? super E> action)	The action specified by <i>action</i> is executed on each unprocessed element in the collection.
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 1-9** The Methods Provided by `ListIterator`

Method	Description
<code>void add(E obj)</code>	Inserts <i>obj</i> into the list in front of the element that will be returned by the next call to <code>next()</code> .
<code>default void forEachRemaining(Consumer<? super E> action)</code>	The action specified by <i>action</i> is executed on each unprocessed element in the collection.
<code>boolean hasNext()</code>	Returns true if there is a next element. Otherwise, returns false .
<code>boolean hasPrevious()</code>	Returns true if there is a previous element. Otherwise, returns false .
<code>E next()</code>	Returns the next element. A NoSuchElementException is thrown if there is not a next element.
<code>int nextIndex()</code>	Returns the index of the next element. If there is not a next element, returns the size of the list.
<code>E previous()</code>	Returns the previous element. A NoSuchElementException is thrown if there is not a previous element.
<code>int previousIndex()</code>	Returns the index of the previous element. If there is not a previous element, returns -1.
<code>void remove()</code>	Removes the current element from the list. An IllegalStateException is thrown if <code>remove()</code> is called before <code>next()</code> or <code>previous()</code> is invoked.
<code>void set(E obj)</code>	Assigns <i>obj</i> to the current element. This is the element last returned by a call to either <code>next()</code> or <code>previous()</code> .

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 added another 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. 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 1-10.



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 <code>Long.MAX_VALUE</code> if the count cannot be obtained for any reason.
default void forEachRemaining(Consumer<? super T> action)	Applies <i>action</i> to each unprocessed element in the data source.
default Comparator<? super T> getComparator()	Returns the comparator used by the invoking spliterator or <code>null</code> if natural ordering is used. If the sequence is unordered, <code>IllegalStateException</code> is thrown.
default long getExactSizeIfKnown()	If the invoking spliterator is sized, returns the number of elements left to iterate. Returns <code>-1</code> otherwise.
default boolean hasCharacteristics(int val)	Returns <code>true</code> if the invoking spliterator has the characteristics passed in <i>val</i> . Returns <code>false</code> otherwise.
boolean tryAdvance(Consumer<? super T> action)	Executes <i>action</i> on the next element in the iteration. Returns <code>true</code> if there is a next element. Returns <code>false</code> 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 <code>null</code> . Thus, if successful, the original spliterator iterates over one portion of the sequence and the returned spliterator iterates over the other portion.

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**. **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. 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 = valsspliterator();
        while(spltitr.tryAdvance((n) -> System.out.println(n)));
        System.out.println();

        // Create new list that contains square roots.
        spltitr = valsspliterator();
        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 = sqrsspliterator();
        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 1-10, 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.

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 1-11](#). 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.

**Table 1-11 The Methods Declared by Map**

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 <i>func</i> to construct a new value. If <i>func</i> 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 <i>func</i> returns null, any preexisting pairing is removed, and null is returned.
default V computeIfAbsent(K k, Function<? super K, ? extends V> func)	Returns the value associated with the key <i>k</i> . Otherwise, the value is constructed through a call to <i>func</i> and the pairing is entered into the map and the constructed value is returned. If no value can be constructed, null is returned.
default V computeIfPresent(K k, BiFunction<? super K, ? super V, ? extends V> func)	If <i>k</i> is in the map, a new value is constructed through a call to <i>func</i> and the new value replaces the old value in the map. In this case, the new value is returned. If the value returned by <i>func</i> is null, the existing key and value are removed from the map and null is returned.
boolean containsKey(Object k)	Returns true if the invoking map contains <i>k</i> as a key. Otherwise, returns false.
boolean containsValue(Object v)	Returns true if the map contains <i>v</i> as a value. Otherwise, returns false.
static <K, V> Map<K, V> copyOf(Map<? extends K, ? extends V> from)	Returns a map that contains the same key/value pairs as that specified by <i>from</i> . The returned map is unmodifiable. Null keys or values are not allowed.
static <K, V> Map.Entry<K, V> entry(K k, V v)	Returns an unmodifiable map entry comprised of the specified key and value. A null key or value is not allowed.
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 <i>obj</i> is a Map and contains the same entries. Otherwise, returns false.
default void forEach(BiConsumer< ? super K, ? super V> action)	Executes <i>action</i> on each element in the invoking map. A ConcurrentModificationException will be thrown if an element is removed during the process.
V get(Object k)	Returns the value associated with the key <i>k</i> . Returns null if the key is not found.
default V getOrDefault(Object k, V defVal)	Returns the value associated with <i>k</i> if it is in the map. Otherwise, <i>defVal</i> is returned.
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.
default V merge(K <i>k</i> , V <i>v</i> , BiFunction<? super K, ? super V, ? extends V> <i>func</i>)	If <i>k</i> is not in the map, the pairing <i>k,v</i> is added to the map. In this case, <i>v</i> is returned. Otherwise, <i>func</i> 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 <i>func</i> is null , the existing key and value are removed from the map and null is returned.
static <K, V> Map<K, V> of(<i>parameter-list</i>)	Creates an unmodifiable map containing the entries specified in <i>parameter-list</i> . Null keys or values are not allowed. Many overloaded versions are provided. See the discussion in the text for details.
static <K, V> Map<K, V> ofEntries(Map.Entry<? extends K, ? extends V> ... <i>entries</i>)	Returns an unmodifiable map that contains the key/value mappings described by the entries passed in <i>entries</i> . Null keys or values are not allowed.
V put(K <i>k</i> , V <i>v</i>)	Puts an entry in the invoking map, overwriting any previous value associated with the key. The key and value are <i>k</i> and <i>v</i> , 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> <i>m</i>)	Puts all the entries from <i>m</i> into this map.
default V putIfAbsent(K <i>k</i> , V <i>v</i>)	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 .
V remove(Object <i>k</i>)	Removes the entry whose key equals <i>k</i> .
default boolean remove(Object <i>k</i> , Object <i>v</i>)	If the key/value pair specified by <i>k</i> and <i>v</i> is in the invoking map, it is removed and true is returned. Otherwise, false is returned.
default boolean replace(K <i>k</i> , V <i>oldV</i> , V <i>newV</i>)	If the key/value pair specified by <i>k</i> and <i>oldV</i> is in the invoking map, the value is replaced by <i>newV</i> and true is returned. Otherwise false is returned.
default V replace(K <i>k</i> , V <i>v</i>)	If the key specified by <i>k</i> is in the invoking map, its value is set to <i>v</i> and the previous value is returned. Otherwise, null is returned.
default void replaceAll(BiFunction< ? super K, ? super V, ? extends V> <i>func</i>)	Executes <i>func</i> on each element of the invoking map, replacing the element with the result returned by <i>func</i> . A ConcurrentModificationException will be thrown if an element is removed during the process.



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.

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.

Beginning with JDK 9, **Map** includes the **of()** factory method, which has a number of overloads. Each version returns an unmodifiable, value-based map that is comprised of the arguments that it is passed. The primary purpose of **of()** is to provide a convenient, efficient way to create a small **Map**. There are 11 overloads of **of()**. One takes no arguments and creates an empty map. It is shown here:

```
static <K, V> Map<K, V> of()
```

Ten overloads take from 1 to 10 arguments and create a list that contains the specified elements. They are shown here:



```
static <K, V> Map<K, V> of(K k1, V v1)
static <K, V> Map<K, V> of(K k1, V v1, K k2, V v2)
static <K, V> Map<K, V> of(K k1, V v1, K k2, V v2, K k3, V v3)
...
static <K, V> Map<K, V> of(K k1, V v1, K k2, V v2, K k3, V v3, K k4, V v4,
                           K k5, V v5, K k6, V v6, K k7, V v7, K k8, V v8,
                           K k9, V v9, K k10, V v10)
```

For all versions, **null** keys and/or values are not allowed. In all cases, the **Map** implementation is unspecified.

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 1-12](#).

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.

**Table 1-12** The Methods Declared by **SortedMap**

Method	Description
Comparator<? super K> comparator()	Returns the invoking sorted map's comparator. If natural ordering is used for the invoking map, <code>null</code> 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 <i>end</i> .
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 <i>start</i> and less than <i>end</i> .
SortedMap<K, V> tailMap(K start)	Returns a map containing those entries with keys that are greater than or equal to <i>start</i> .

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 1-13. 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.

**Table 1-13 The Methods Declared by NavigableMap**

Method	Description
Map.Entry<K,V> ceilingEntry(K <i>obj</i>)	Searches the map for the smallest key <i>k</i> such that $k \geq obj$. If such a key is found, its entry is returned. Otherwise, null is returned.
K ceilingKey(K <i>obj</i>)	Searches the map for the smallest key <i>k</i> such that $k \geq obj$. 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 <i>obj</i>)	Searches the map for the largest key <i>k</i> such that $k \leq obj$. If such a key is found, its entry is returned. Otherwise, null is returned.
K floorKey(K <i>obj</i>)	Searches the map for the largest key <i>k</i> such that $k \leq obj$. If such a key is found, it is returned. Otherwise, null is returned.
NavigableMap<K,V> headMap(K <i>upperBound</i> , boolean <i>incl</i>)	Returns a NavigableMap that includes all entries from the invoking map that have keys that are less than <i>upperBound</i> . If <i>incl</i> is true , then an element equal to <i>upperBound</i> is included. The resulting map is backed by the invoking map.
Map.Entry<K,V> higherEntry(K <i>obj</i>)	Searches the set for the largest key <i>k</i> such that $k > obj$. If such a key is found, its entry is returned. Otherwise, null is returned.
K higherKey(K <i>obj</i>)	Searches the set for the largest key <i>k</i> 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 <i>obj</i>)	Searches the set for the largest key <i>k</i> such that $k < obj$. If such a key is found, its entry is returned. Otherwise, null is returned.
K lowerKey(K <i>obj</i>)	Searches the set for the largest key <i>k</i> such that $k < obj$. If such a key is found, it is returned. Otherwise, null is returned.



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 <i>lowerBound</i> , boolean <i>lowIncl</i> , K <i>upperBound</i> boolean <i>highIncl</i>)	Returns a NavigableMap that includes all entries from the invoking map that have keys that are greater than <i>lowerBound</i> and less than <i>upperBound</i> . If <i>lowIncl</i> is true , then an element equal to <i>lowerBound</i> is included. If <i>highIncl</i> is true , then an element equal to <i>highIncl</i> is included. The resulting map is backed by the invoking map.
NavigableMap<K,V> tailMap(K <i>lowerBound</i> , boolean <i>incl</i>)	Returns a NavigableMap that includes all entries from the invoking map that have keys that are greater than <i>lowerBound</i> . If <i>incl</i> is true , then an element equal to <i>lowerBound</i> is included. The resulting map is backed by the invoking map.

The Map.Entry Interface

The **Map.Entry** interface enables you to work with a map entry. For example, 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 1-14 summarizes the non-static methods declared by **Map.Entry**. It also has 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.

**Table 1-14** The Non-Static Methods Declared by `Map.Entry`

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

The Map Classes

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

Class	Description
<code>AbstractMap</code>	Implements most of the <code>Map</code> interface.
<code>EnumMap</code>	Extends <code>AbstractMap</code> for use with <code>enum</code> keys.
<code>HashMap</code>	Extends <code>AbstractMap</code> to use a hash table.
<code>TreeMap</code>	Extends <code>AbstractMap</code> to use a tree.
<code>WeakHashMap</code>	Extends <code>AbstractMap</code> to use a hash table with weak keys.
<code>LinkedHashMap</code>	Extends <code>HashMap</code> to allow insertion-order iterations.
<code>IdentityHashMap</code>	Extends <code>AbstractMap</code> 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", 3434.34);  
        hm.put("Tom Smith", 123.22);  
        hm.put("Jane Baker", 1378.00);  
        hm.put("Tod Hall", 99.22);  
        hm.put("Ralph Smith", -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", 3434.34);  
        tm.put("Tom Smith", 123.22);  
        tm.put("Jane Baker", 1378.00);  
        tm.put("Tod Hall", 99.22);  
        tm.put("Ralph Smith", -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 dramatically changed. JDK 8 added 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 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>
    thenComparingIntToIntFunction<? 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, **Comparator** has 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>
    comparingIntToIntFunction<? 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.) 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, 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.

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", 3434.34);  
        tm.put("Tom Smith", 123.22);  
        tm.put("Jane Baker", 1378.00);  
        tm.put("Tod Hall", 99.22);  
        tm.put("Ralph Smith", -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.

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", 3434.34);
    }
}
```



```
tm.put("Tom Smith", 123.22);
tm.put("Jane Baker", 1378.00);
tm.put("Tod Hall", 99.22);
tm.put("Ralph Smith", -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 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.

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 1-15.

**Table 1-15** The Algorithms Defined by **Collections**

Method	Description
static <T> boolean addAll(Collection <? super T> c, T... elements)	Inserts the elements specified by <i>elements</i> into the collection specified by <i>c</i> . 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 <i>c</i> .
static <T> int binarySearch(List<? extends T> list, T value, Comparator<? super T> c)	Searches for <i>value</i> in <i>list</i> ordered according to <i>c</i> . Returns the position of <i>value</i> in <i>list</i> , or a negative value if <i>value</i> is not found.
static <T> int binarySearch(List<? extends Comparable<? super T>> list, T value)	Searches for <i>value</i> in <i>list</i> . The list must be sorted. Returns the position of <i>value</i> in <i>list</i> , or a negative value if <i>value</i> 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 .
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 .
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 .



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 .
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 <i>list2</i> to <i>list1</i> .
static boolean disjoint(Collection<?> a, Collection<?> b)	Compares the elements in <i>a</i> to elements in <i>b</i> . 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.
static <E> NavigableSet<E> emptyNavigableSet()	Returns an immutable, empty NavigableSet object of the inferred type.
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.
static <E> SortedSet<E> emptySortedSet()	Returns an immutable, empty SortedSet object of the inferred type.
static <T> Enumeration<T> enumeration(Collection<T> c)	Returns an enumeration over <i>c</i> . (See "The Enumeration Interface," later in this chapter.)
static <T> void fill(List<? super T> list, T obj)	Assigns <i>obj</i> to each element of <i>list</i> .



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



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



static <K, V> NavigableMap<K, V> unmodifiableNavigableMap(NavigableMap<K, ? extends V> nm)	Returns an unmodifiable navigable map backed by <i>nm</i> .
static <T> NavigableSet<T> unmodifiableNavigableSet(NavigableSet<T> ns)	Returns an unmodifiable navigable set backed by <i>ns</i> .
static <T> Set<T> unmodifiableSet(Set<? extends T> s)	Returns an unmodifiable set backed by <i>s</i> .
static <K, V> SortedMap<K, V> unmodifiableSortedMap(SortedMap<K, ? extends V> sm)	Returns an unmodifiable sorted map backed by <i>sm</i> .
static <T> SortedSet<T> unmodifiableSortedSet(SortedSet<T> ss)	Returns an unmodifiable sorted set backed by <i>ss</i> .

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.

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), **nulls** (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), **nulls** (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**. Here are a number of its forms. Several more versions are available that let you specify a range and/or a comparator.

```
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.

One quite powerful method in **Arrays** 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.

Arrays supports 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.

Arrays supports the **Stream** interface 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.

Another two methods 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.

One of the more intriguing methods defined by **Arrays** 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.

JDK 9 added three comparison methods to **Arrays**. They are **compare()**, **compareUnsigned()**, and **mismatch()**. Each has several overloads and each has versions that let you define a range to compare. Here is a brief description of each. The **compare()** method compares two arrays. It returns zero if they are the same, a positive value if the first array is greater than the second, and negative if the first array is less than the second. To perform an unsigned comparison of two arrays that hold integer values, use **compareUnsigned()**. To find the location of the first mismatch between two arrays, use **mismatch()**. It returns the index of the mismatch, or -1 if the arrays are equivalent.

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.

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. 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 abstract 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.

JDK 9 added a default method to **Enumeration** called **asIterator()**. It is shown here:

```
default Iterator<E> asIterator( )
```

It returns an iterator to the elements in the enumeration. As such, it provides an easy way to convert an old-style **Enumeration** into a modern **Iterator**.



Furthermore, if a portion of the elements in the enumeration have already been read prior to calling **asIterator()**, the returned iterator accesses only the remaining elements.

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 1-16.

**Table 1-16** The Legacy Methods Defined by **Vector**

Method	Description
void addElement(E <i>element</i>)	The object specified by <i>element</i> 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 <i>element</i>)	Returns true if <i>element</i> is contained by the vector, and returns false if it is not.
void copyInto(Object <i>array</i> [])	The elements contained in the invoking vector are copied into the array specified by <i>array</i> .
E elementAt(int <i>index</i>)	Returns the element at the location specified by <i>index</i> .
Enumeration<E> elements()	Returns an enumeration of the elements in the vector.
void ensureCapacity(int <i>size</i>)	Sets the minimum capacity of the vector to <i>size</i> .
E firstElement()	Returns the first element in the vector.
int indexOf(Object <i>element</i>)	Returns the index of the first occurrence of <i>element</i> . If the object is not in the vector, -1 is returned.
int indexOf(Object <i>element</i> , int <i>start</i>)	Returns the index of the first occurrence of <i>element</i> at or after <i>start</i> . If the object is not in that portion of the vector, -1 is returned.



void insertElementAt(E <i>element</i> , int <i>index</i>)	Adds <i>element</i> to the vector at the location specified by <i>index</i> .
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 <i>element</i>)	Returns the index of the last occurrence of <i>element</i> . If the object is not in the vector, -1 is returned.
int lastIndexOf(Object <i>element</i> , int <i>start</i>)	Returns the index of the last occurrence of <i>element</i> before <i>start</i> . 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 <i>element</i>)	Removes <i>element</i> 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 <i>index</i>)	Removes the element at the location specified by <i>index</i> .
void setElementAt(E <i>element</i> , int <i>index</i>)	The location specified by <i>index</i> is assigned <i>element</i> .
void setSize(int <i>size</i>)	Sets the number of elements in the vector to <i>size</i> . 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.

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

The following program uses a vector to store various types of numeric



objects. It demonstrates several of the legacy methods defined by **Vector**. It also demonstrates the **Enumeration** interface.

```
// Demonstrate various Vector operations.  
import java.util.*;  
  
class VectorDemo {  
    public static void main(String args[]) {  
  
        // initial size is 3, increment is 2  
        Vector<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();
}
}
```



The output from this program is shown here:

```
Initial size: 0
Initial capacity: 3
Capacity after four additions: 5
Current capacity: 5
Current capacity: 7
Current capacity: 9
First element: 1
Last element: 12
Vector contains 3.
```

```
Elements in vector:
1 2 3 4 5 6 7 9 10 11 12
```

Instead of relying on an enumeration to cycle through the objects (as the preceding program does), you can use an iterator. For example, the following iterator-based code can be substituted into the program:

```
// Use an iterator to display contents.
Iterator<Integer> vItr = v.iterator();

System.out.println("\nElements in vector:");
while(vItr.hasNext())
    System.out.print(vItr.next() + " ");
System.out.println();
```

You can also use a for-each **for** loop to cycle through a **Vector**, as the following version of the preceding code shows:



```
// Use an enhanced for loop to display contents
System.out.println("\nElements in vector:");
for(int i : v)
    System.out.print(i + " ");
System.out.println();
```

Because the **Enumeration** interface is not recommended for new code, you will usually use an iterator or a for-each **for** loop to enumerate the contents of a vector. Of course, legacy code will employ **Enumeration**. Fortunately, enumerations and iterators work in nearly the same manner.

Stack

Stack is a subclass of **Vector** that implements a standard last-in, first-out stack. **Stack** only defines the default constructor, which creates an empty stack. With the release of JDK 5, **Stack** was retrofitted for generics and is declared as shown here:

```
class Stack<E>
```

Here, **E** specifies the type of element stored in the stack.

Stack includes all the methods defined by **Vector** and adds several of its own, shown in Table 1-17.

**Table 1-17** The Methods Defined by Stack

Method	Description
boolean empty()	Returns true if the stack is empty, and returns false if the stack contains elements.
E peek()	Returns the element on the top of the stack, but does not remove it.
E pop()	Returns the element on the top of the stack, removing it in the process.
E push(E <i>element</i>)	Pushes <i>element</i> onto the stack. <i>element</i> is also returned.
int search(Object <i>element</i>)	Searches for <i>element</i> in the stack. If found, its offset from the top of the stack is returned. Otherwise, -1 is returned.

To put an object on the top of the stack, call **push()**. To remove and return the top element, call **pop()**. You can use **peek()** to return, but not remove, the top object. An **EmptyStackException** is thrown if you call **pop()** or **peek()** when the invoking stack is empty. The **empty()** method returns **true** if nothing is on the stack. The **search()** method determines whether an object exists on the stack and returns the number of pops that are required to bring it to the top of the stack. Here is an example that creates a stack, pushes several **Integer** objects onto it, and then pops them off again:



```
// Demonstrate the Stack class.  
import java.util.*;  
  
class StackDemo {  
    static void showpush(Stack<Integer> st, int a) {  
        st.push(a);  
        System.out.println("push(" + a + ")");  
        System.out.println("stack: " + st);  
    }  
  
    static void showpop(Stack<Integer> st) {  
        System.out.print("pop -> ");  
        Integer a = st.pop();  
        System.out.println(a);  
        System.out.println("stack: " + st);  
    }  
  
    public static void main(String args[]) {  
        Stack<Integer> st = new Stack<Integer>();  
  
        System.out.println("stack: " + st);  
        showpush(st, 42);  
        showpush(st, 66);  
        showpush(st, 99);  
        showpop(st);  
        showpop(st);  
        showpop(st);  
  
        try {  
            showpop(st);  
        } catch (EmptyStackException e) {  
            System.out.println("empty stack");  
        }  
    }  
}
```

The following is the output produced by the program; notice how the exception handler for **EmptyStackException** is used so that you can gracefully handle a stack underflow:



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

One other point: although **Stack** is not deprecated, **ArrayDeque** is a better choice.

Dictionary

Dictionary is an abstract class that represents a key/value storage repository and operates much like **Map**. Given a key and value, you can store the value in a **Dictionary** object. Once the value is stored, you can retrieve it by using its key. Thus, like a map, a dictionary can be thought of as a list of key/value pairs. Although not currently deprecated, **Dictionary** is classified as obsolete, because it is fully superseded by **Map**. However, **Dictionary** is still in use and thus is discussed here.

With the advent of JDK 5, **Dictionary** was made generic. It is declared as shown here:

```
class Dictionary<K, V>
```

Here, **K** specifies the type of keys, and **V** specifies the type of values. The abstract methods defined by **Dictionary** are listed in Table 1-18.

**Table 1-18** The Abstract Methods Defined by **Dictionary**

Method	Purpose
Enumeration<V> elements()	Returns an enumeration of the values contained in the dictionary.
V get(Object key)	Returns the object that contains the value associated with <i>key</i> . If <i>key</i> is not in the dictionary, a null object is returned.
boolean isEmpty()	Returns true if the dictionary is empty, and returns false if it contains at least one key.
Enumeration<K> keys()	Returns an enumeration of the keys contained in the dictionary.
V put(K key, V value)	Inserts a key and its value into the dictionary. Returns null if <i>key</i> is not already in the dictionary; returns the previous value associated with <i>key</i> if <i>key</i> is already in the dictionary.
V remove(Object key)	Removes <i>key</i> and its value. Returns the value associated with <i>key</i> . If <i>key</i> is not in the dictionary, a null is returned.
int size()	Returns the number of entries in the dictionary.

To add a key and a value, use the **put()** method. Use **get()** to retrieve the value of a given key. The keys and values can each be returned as an **Enumeration** by the **keys()** and **elements()** methods, respectively. The **size()** method returns the number of key/value pairs stored in a dictionary, and **isEmpty()** returns **true** when the dictionary is empty. You can use the **remove()** method to delete a key/value pair.

Hashtable

Hashtable was part of the original **java.util** and is a concrete implementation of a **Dictionary**. However, with the advent of collections, **Hashtable** was reengineered to also implement the **Map** interface. Thus, **Hashtable** is integrated into the Collections Framework. It is similar to **HashMap**, but is synchronized.

Like **HashMap**, **Hashtable** stores key/value pairs in a hash table. However, neither keys nor values can be **null**. When using a **Hashtable**, you specify an object that is used as a key, and the value that you want linked to that key. The key is then hashed, and the resulting hash code is used as the index at which



the value is stored within the table.

Hashtable was made generic by JDK 5. It is declared like this:

```
class Hashtable<K, V>
```

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

A hash table can only store keys that override the **hashCode()** and **equals()** methods that are defined by **Object**. The **hashCode()** method must compute and return the hash code for the object. Of course, **equals()** compares two objects. Fortunately, many of Java's built-in classes already implement the **hashCode()** method. For example, the most common type of **Hashtable** uses a **String** object as the key. **String** implements both **hashCode()** and **equals()**.

The **Hashtable** constructors are shown here:

```
Hashtable()
Hashtable(int size)
Hashtable(int size, float fillRatio)
Hashtable(Map<? extends K, ? extends V> m)
```

The first version is the default constructor. The second version creates a hash table that has an initial size specified by *size*. (The default size is 11.) The third version creates a hash table that has an initial size specified by *size* and a fill ratio specified by *fillRatio*. This ratio (also referred to as a *load factor*) must be between 0.0 and 1.0, and it determines how full the hash table can be before it is resized upward. Specifically, when the number of elements is greater than the capacity of the hash table multiplied by its fill ratio, the hash table is expanded. If you do not specify a fill ratio, then 0.75 is used. Finally, the fourth version creates a hash table that is initialized with the elements in *m*. The default load factor of 0.75 is used.

In addition to the methods defined by the **Map** interface, which **Hashtable** now implements, **Hashtable** defines the legacy methods listed in Table 1-19. Several methods throw **NullPointerException** if an attempt is made to use a **null** key or value.

**Table 1-19** The Legacy Methods Defined by **Hashtable**

Method	Description
void clear()	Resets and empties the hash table.
Object clone()	Returns a duplicate of the invoking object.
boolean contains(Object <i>value</i>)	Returns true if some value equal to <i>value</i> exists within the hash table. Returns false if the value isn't found.
boolean containsKey(Object <i>key</i>)	Returns true if some key equal to <i>key</i> exists within the hash table. Returns false if the key isn't found.
boolean containsValue(Object <i>value</i>)	Returns true if some value equal to <i>value</i> exists within the hash table. Returns false if the value isn't found.
Enumeration<V> elements()	Returns an enumeration of the values contained in the hash table.
V get(Object <i>key</i>)	Returns the object that contains the value associated with <i>key</i> . If <i>key</i> is not in the hash table, a null object is returned.
boolean isEmpty()	Returns true if the hash table is empty; returns false if it contains at least one key.
Enumeration<K> keys()	Returns an enumeration of the keys contained in the hash table.
V put(K <i>key</i> , V <i>value</i>)	Inserts a key and a value into the hash table. Returns null if <i>key</i> isn't already in the hash table; returns the previous value associated with <i>key</i> if <i>key</i> is already in the hash table.
void rehash()	Increases the size of the hash table and rehashes all of its keys.
V remove(Object <i>key</i>)	Removes <i>key</i> and its value. Returns the value associated with <i>key</i> . If <i>key</i> is not in the hash table, a null object is returned.
int size()	Returns the number of entries in the hash table.
String toString()	Returns the string equivalent of a hash table.

The following example reworks the bank account program, shown earlier, so that it uses a **Hashtable** to store the names of bank depositors and their current balances:



```
// Demonstrate a Hashtable.  
import java.util.*;  
  
class HTDemo {  
    public static void main(String args[]) {  
        Hashtable<String, Double> balance =  
            new Hashtable<String, Double>();  
  
        Enumeration<String> names;  
        String str;  
        double bal;  
  
        balance.put("John Doe", 3434.34);  
        balance.put("Tom Smith", 123.22);  
  
        balance.put("Jane Baker", 1378.00);  
        balance.put("Tod Hall", 99.22);  
        balance.put("Ralph Smith", -19.08);  
  
        // Show all balances in hashtable.  
        names = balance.keys();  
        while(names.hasMoreElements()) {  
            str = names.nextElement();  
            System.out.println(str + ": " +  
                balance.get(str));  
        }  
  
        System.out.println();  
  
        // Deposit 1,000 into John Doe's account.  
        bal = balance.get("John Doe");  
        balance.put("John Doe", bal+1000);  
        System.out.println("John Doe's new balance: " +  
            balance.get("John Doe"));  
    }  
}
```



The output from this program is shown here:

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

John Doe's new balance: 4434.34
```

One important point: Like the map classes, **Hashtable** does not directly support iterators. Thus, the preceding program uses an enumeration to display the contents of **balance**. However, you can obtain set-views of the hash table, which permits the use of iterators. To do so, you simply use one of the collection-view methods defined by **Map**, such as **entrySet()** or **keySet()**. For example, you can obtain a set-view of the keys and cycle through them using either an iterator or an enhanced **for** loop. Here is a reworked version of the program that shows this technique:

```
// Use iterators with a Hashtable.
import java.util.*;

class HTDemo2 {
    public static void main(String args[]) {
        Hashtable<String, Double> balance =
            new Hashtable<String, Double>();

        String str;
        double bal;

        balance.put("John Doe", 3434.34);
        balance.put("Tom Smith", 123.22);
```



```
balance.put("Jane Baker", 1378.00);
balance.put("Tod Hall", 99.22);
balance.put("Ralph Smith", -19.08);

// Show all balances in hashtable.
// First, get a set view of the keys.
Set<String> set = balance.keySet();

// Get an iterator.
Iterator<String> itr = set.iterator();
while(itr.hasNext()) {
    str = itr.next();
    System.out.println(str + ": " +
                        balance.get(str));
}

System.out.println();

// Deposit 1,000 into John Doe's account.
bal = balance.get("John Doe");
balance.put("John Doe", bal+1000);
System.out.println("John Doe's new balance: " +
                    balance.get("John Doe"));
}
```

Properties

Properties is a subclass of **Hashtable**. It is used to maintain lists of values in which the key is a **String** and the value is also a **String**. The **Properties** class is used by some other Java classes. For example, it is the type of object returned by **System.getProperties()** when obtaining environmental values. Although the **Properties** class, itself, is not generic, several of its methods are.

Properties defines the following protected volatile instance variable:
Properties defaults;
This variable holds a default property list associated with a **Properties** object.



Properties defines these constructors:

```
Properties( )
Properties(Properties propDefault)
Properties(int capacity)
```

The first version creates a **Properties** object that has no default values. The second creates an object that uses *propDefault* for its default values. In both cases, the property list is empty. The third constructor lets you specify an initial capacity for the property list. In all cases, the list will grow as needed.

In addition to the methods that **Properties** inherits from **Hashtable**, **Properties** defines the methods listed in Table 1-20. **Properties** also contains one deprecated method: **save()**. This was replaced by **store()** because **save()** did not handle errors correctly.

**Table 1-20 The Methods Defined by Properties**

Method	Description
<code>String getProperty(String key)</code>	Returns the value associated with <i>key</i> . A <code>null</code> object is returned if <i>key</i> is neither in the list nor in the default property list.
<code>String getProperty(String key, String defaultProperty)</code>	Returns the value associated with <i>key</i> . <i>defaultProperty</i> is returned if <i>key</i> is neither in the list nor in the default property list.
<code>void list(PrintStream streamOut)</code>	Sends the property list to the output stream linked to <i>streamOut</i> .
<code>void list(PrintWriter streamOut)</code>	Sends the property list to the output stream linked to <i>streamOut</i> .
<code>void load(InputStream streamIn) throws IOException</code>	Inputs a property list from the input stream linked to <i>streamIn</i> .
<code>void load(Reader streamIn) throws IOException</code>	Inputs a property list from the input stream linked to <i>streamIn</i> .
<code>void loadFromXML(InputStream streamIn) throws IOException, InvalidPropertiesFormatException</code>	Inputs a property list from an XML document linked to <i>streamIn</i> .
<code>Enumeration<?> propertyNames()</code>	Returns an enumeration of the keys. This includes those keys found in the default property list, too.
<code>Object setProperty(String key, String value)</code>	Associates <i>value</i> with <i>key</i> . Returns the previous value associated with <i>key</i> , or returns <code>null</code> if no such association exists.
<code>void store(OutputStream streamOut, String description) throws IOException</code>	After writing the string specified by <i>description</i> , the property list is written to the output stream <i>streamOut</i> .
<code>void store(Writer streamOut, String description) throws IOException</code>	After writing the string specified by <i>description</i> , the property list is written to the output stream <i>streamOut</i> .
<code>void storeToXML(OutputStream streamOut, String description) throws IOException</code>	The property list and the string specified by <i>description</i> is written as an XML document to <i>streamOut</i> .
<code>void storeToXML(OutputStream streamOut, String description, String enc)</code>	The property list and the string specified by <i>description</i> is written as an XML document to <i>streamOut</i> using the specified character encoding.
<code>void storeToXML(OutputStream streamOut, String description, Charset cs)</code>	The property list and the string specified by <i>description</i> is written as an XML document to <i>streamOut</i> using the specified encoding.
<code>Set<String> stringPropertyNames()</code>	Returns a set of keys.



One useful capability of the **Properties** class is that you can specify a default property that will be returned if no value is associated with a certain key. For example, a default value can be specified along with the key in the `getProperty()` method—such as `getProperty("name","default value")`. If the "name" value is not found, then "default value" is returned. When you construct a **Properties** object, you can pass another instance of **Properties** to be used as the default properties for the new instance. In this case, if you call `getProperty("foo")` on a given **Properties** object, and "foo" does not exist, Java looks for "foo" in the default **Properties** object. This allows for arbitrary nesting of levels of default properties.

The following example demonstrates **Properties**. It creates a property list in which the keys are the names of states and the values are the names of their capitals. Notice that the attempt to find the capital for Florida includes a default value.



```
// Demonstrate a Property list.  
import java.util.*;  
  
class PropDemo {  
    public static void main(String args[]) {  
        Properties capitals = new Properties();  
  
        capitals.setProperty("Illinois", "Springfield");  
        capitals.setProperty("Missouri", "Jefferson City");  
        capitals.setProperty("Washington", "Olympia");  
        capitals.setProperty("California", "Sacramento");  
        capitals.setProperty("Indiana", "Indianapolis");  
  
        // Get a set-view of the keys.  
        Set<?> states = capitals.keySet();  
  
        // Show all of the states and capitals.  
        for(Object name : states)  
            System.out.println("The capital of " +  
                               name + " is " +  
                               capitals.getProperty((String)name)  
                               + ".");  
  
        System.out.println();  
  
        // Look for state not in list -- specify default.  
        String str = capitals.getProperty("Florida", "Not Found");  
        System.out.println("The capital of Florida is " + str + ".");  
    }  
}
```

The output from this program is shown here:



The capital of Missouri is Jefferson City.
The capital of Illinois is Springfield.
The capital of Indiana is Indianapolis.
The capital of California is Sacramento.
The capital of Washington is Olympia.

The capital of Florida is Not Found.

Since Florida is not in the list, the default value is used.

Although it is perfectly valid to use a default value when you call **getProperty()**, as the preceding example shows, there is a better way of handling default values for most applications of property lists. For greater flexibility, specify a default property list when constructing a **Properties** object. The default list will be searched if the desired key is not found in the main list. For example, the following is a slightly reworked version of the preceding program, with a default list of states specified. Now, when Florida is sought, it will be found in the default list:



```
// Use a default property list.  
import java.util.*;  
  
class PropDemoDef {  
    public static void main(String args[]) {  
        Properties defList = new Properties();  
        defList.setProperty("Florida", "Tallahassee");  
        defList.setProperty("Wisconsin", "Madison");  
  
        Properties capitals = new Properties(defList);  
  
        capitals.setProperty("Illinois", "Springfield");  
        capitals.setProperty("Missouri", "Jefferson City");  
        capitals.setProperty("Washington", "Olympia");  
        capitals.setProperty("California", "Sacramento");  
        capitals.setProperty("Indiana", "Indianapolis");  
  
        // Get a set-view of the keys.  
        Set<?> states = capitals.keySet();  
  
        // Show all of the states and capitals.  
        for(Object name : states)  
            System.out.println("The capital of " +  
                               name + " is " +  
                               capitals.getProperty((String)name)  
                               + ".");  
  
        System.out.println();  
  
        // Florida will now be found in the default list.  
        String str = capitals.getProperty("Florida");  
        System.out.println("The capital of Florida is "  
                           + str + ".");  
    }  
}
```



Using `store()` and `load()`

One of the most useful aspects of **Properties** is that the information contained in a **Properties** object can be easily stored to or loaded from disk with the **store()** and **load()** methods. At any time, you can write a **Properties** object to a stream or read it back. This makes property lists especially convenient for implementing simple databases. For example, the following program uses a property list to create a simple computerized telephone book that stores names and phone numbers. To find a person's number, you enter his or her name. The program uses the **store()** and **load()** methods to store and retrieve the list. When the program executes, it first tries to load the list from a file called **phonebook.dat**. If this file exists, the list is loaded. You can then add to the list. If you do, the new list is saved when you terminate the program. Notice how little code is required to implement a small, but functional, computerized phone book.



```
/* A simple telephone number database that uses
   a property list. */
import java.io.*;
import java.util.*;

class Phonebook {
    public static void main(String args[])
        throws IOException
    {
        Properties ht = new Properties();
        BufferedReader br =
            new BufferedReader(new InputStreamReader(System.in));
        String name, number;
        FileInputStream fin = null;
        boolean changed = false;

        // Try to open phonebook.dat file.
        try {
            fin = new FileInputStream("phonebook.dat");
        } catch(FileNotFoundException e) {
            // ignore missing file
        }

        /* If phonebook file already exists,
           load existing telephone numbers. */
        try {
            if(fin != null) {
                ht.load(fin);
                fin.close();
            }
        } catch(IOException e) {
            System.out.println("Error reading file.");
        }

        // Let user enter new names and numbers.
        do {
            System.out.println("Enter new name" +
                               " ('quit' to stop): ");
            name = br.readLine();
            if(name.equals("quit")) continue;

            System.out.println("Enter number: ");
            number = br.readLine();

            ht.setProperty(name, number);
            changed = true;
        } while(!name.equals("quit"));
    }
}
```



```
// If phone book data has changed, save it.  
if(changed) {  
    FileOutputStream fout = new FileOutputStream("phonebook.dat");  
  
    ht.store(fout, "Telephone Book");  
    fout.close();  
}  
  
// Look up numbers given a name.  
do {  
    System.out.println("Enter name to find" +  
                       " ('quit' to quit): ");  
    name = br.readLine();  
    if(name.equals("quit")) continue;  
  
    number = (String) ht.get(name);  
    System.out.println(number);  
} while(!name.equals("quit"));  
}  
}
```

Parting Thoughts on Collections

The Collections Framework gives you, the programmer, a powerful set of well-engineered solutions to some of programming's most common tasks. Consider using a collection the next time you need to store and retrieve information. Remember, collections need not be reserved for only the “large jobs,” such as corporate databases, mailing lists, or inventory systems. They are also effective when applied to smaller jobs. For example, a **TreeMap** might make an excellent collection to hold the directory structure of a set of files. A **TreeSet** could be quite useful for storing project-management information.