CHAPTER

18 java.util Part 1: The Collections Framework

This chapter begins our examination of **java.util**. This important package contains a large assortment of classes and interfaces that support a broad range of functionality. For example, **java.util** has classes that generate pseudorandom numbers, manage date and time, observe events, manipulate sets of bits, tokenize strings, and handle formatted data. The **java.util** package also contains one of Java's most powerful subsystems: the *Collections Framework*. The Collections Framework is a sophisticated hierarchy of interfaces and classes that provide state-of-the-art technology for managing groups of objects. It merits close attention by all programmers.

Because **java.util** contains a wide array of functionality, it is quite large. Here is a list of its top-level classes:

AbstractCollection	FormattableFlags	Properties
AbstractList	Formatter	PropertyPermission
AbstractMap	GregorianCalendar	PropertyResourceBundle
AbstractQueue	HashMap	Random
AbstractSequentialList	HashSet	ResourceBundle
AbstractSet	Hashtable	Scanner
ArrayDeque	IdentityHashMap	ServiceLoader
ArrayList	IntSummaryStatistics (Added by JDK 8.)	SimpleTimeZone
Arrays	LinkedHashMap	Spliterators (Added by JDK 8.)
Base64 (Added by JDK 8.)	LinkedHashSet	SplitableRandom (Added by JDK 8.)
BitSet	LinkedList	Stack
Calendar	ListResourceBundle	StringJoiner (Added by JDK 8.)

Collections	Locale	StringTokenizer
Currency	LongSummaryStatistics (Added by JDK 8.)	Timer
Date	Objects	TimerTask
Dictionary	Observable	TimeZone
DoubleSummaryStatistics (Added by JDK 8.)	Optional (Added by JDK 8.)	TreeMap
EnumMap	OptionalDouble (Added by JDK 8.)	TreeSet
EnumSet	OptionalInt (Added by JDK 8.)	UUID
EventListenerProxy	OptionalLong (Added by JDK 8.)	Vector
EventObject	PriorityQueue	WeakHashMap

The interfaces defined by **java.util** are shown next:

Collection	Map.Entry	Set
Comparator	NavigableMap	SortedMap
Deque	NavigableSet	SortedSet
Enumeration	Observer	Spliterator (Added by JDK 8.)
EventListener	PrimitiveIterator (Added by JDK 8.)	Spliterator.OfDouble (Added by JDK 8.)
Formattable	PrimitiveIterator.OfDouble (Added by JDK 8.)	Spliterator.OfInt (Added by JDK 8.)
Iterator	PrimitiveIterator.OfInt (Added by JDK 8.)	Spliterator.OfLong (Added by JDK 8.)
List	PrimitiveIterator.OfLong (Added by JDK 8.)	Spliterator.OfPrimitive (Added by JDK 8.)
ListIterator	Queue	
Map	RandomAccess	

Because of its size, the description of **java.util** is broken into two chapters. This chapter examines those members of **java.util** that are part of the Collections Framework. Chapter 18 discusses its other classes and interfaces.

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 are an answer to these (and other) problems.

The Collections Framework was designed to meet several goals. First, the framework had to be high-performance. The implementations for the fundamental collections (dynamic arrays, linked lists, trees, and hash tables) are highly efficient. You seldom, if ever, need to code one of these "data engines" manually. Second, the framework had to allow different types of collections to work in a similar manner and with a high degree of interoperability. Third, extending and/or adapting a collection had to be easy. Toward this end, the entire Collections Framework is 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 adds 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 adds 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.

NOTE If you are familiar with C++, then you will find it helpful to know that the Java collections technology is similar in spirit to the Standard Template Library (STL) defined by C++. What C++ calls a container, Java calls a collection. However, there are significant differences between the Collections Framework and the STL. It is important to not jump to conclusions.

JDK 5 Changed the Collections Framework

When JDK 5 was released, some fundamental changes were made to the Collections Framework that significantly increased its power and streamlined its use. These changes include the addition of generics, autoboxing/unboxing, and the for-each style **for** loop. Although JDK 8 is three major Java releases after JDK 5, the effects of the JDK 5 features were so profound that they still warrant special attention. The main reason is that you may encounter pre-JDK 5 code. Understanding the effects and reasons for the changes is important if you will be maintaining or updating older code.

Generics Fundamentally Changed the Collections Framework

The addition of generics caused a significant change to the Collections Framework because the entire Collections Framework was reengineered for it. All collections are now generic, and many of the methods that operate on collections take generic type parameters. Simply put, the addition of generics affected every part of the Collections Framework.

Generics added the one feature that collections had been missing: type safety. Prior to generics, all collections stored **Object** references, which meant that any collection could store any type of object. Thus, it was possible to accidentally store incompatible types in a collection. Doing so could result in run-time type mismatch errors. With generics, it is possible to explicitly state the type of data being stored, and run-time type mismatch errors can be avoided.

Although the addition of generics changed the declarations of most of its classes and interfaces, and several of their methods, overall, the Collections Framework still works the same as it did prior to generics. Of course, to gain the advantages that generics bring collections, older code will need to be rewritten. This is also important because pre-generics code will generate warning messages when compiled by a modern Java compiler. To eliminate these warnings, you will need to add type information to all your collections code.

Autoboxing Facilitates the Use of Primitive Types

Autoboxing/unboxing facilitates the storing of primitive types in collections. As you will see, a collection can store only references, not primitive values. In the past, if you wanted to store a primitive value, such as an **int**, in a collection, you had to manually box it into its type wrapper. When the value was retrieved, it needed to be manually unboxed (by using an explicit cast) into its proper primitive type. Because of autoboxing/unboxing, Java can automatically perform the proper boxing and unboxing needed when storing or retrieving primitive types. There is no need to manually perform these operations.

The For-Each Style for Loop

All collection classes in the Collections Framework were retrofitted to implement the **Iterable** interface, which means that a collection can be cycled through by use of the for-each style **for** loop. In the past, cycling through a collection required the use of an iterator (described later in this chapter), with the programmer manually constructing the loop. Although iterators are still needed for some uses, in many cases, iterator-based loops can be replaced by **for** loops.

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**, and **ListIterator** interfaces, which are described in depth later in this chapter. Beginning with JDK 8, **Spliterator** can also be used. 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 following sections examine the collection interfaces.

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 18-1. 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.

Method	Description
boolean add(E *obj*)	Adds *obj* to the invoking collection. Returns **true** if *obj* was added to the collection. Returns **false** if *obj* is already a member of the collection and the collection does not allow duplicates.
boolean addAll(Collection<? extends E> *c*)	Adds all the elements of *c* 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 *obj*)	Returns **true** if *obj* is an element of the invoking collection. Otherwise, returns **false**.
boolean containsAll(Collection<?> *c*)	Returns **true** if the invoking collection contains all elements of *c*. Otherwise, returns **false**.
boolean equals(Object *obj*)	Returns **true** if the invoking collection and *obj* are equal. Otherwise, returns **false**.
int hashCode( )	Returns the hash code for the invoking collection.
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. (Added by JDK 8.)
boolean remove(Object *obj*)	Removes one instance of *obj* from the invoking collection. Returns **true** if the element was removed. Otherwise, returns **false**.
boolean removeAll(Collection<?> *c*)	Removes all elements of *c* from the invoking collection. Returns **true** if the collection changed (i.e., elements were removed). Otherwise, returns **false**.
default boolean removeIf(Predicate<? super E> *predicate*)	Removes from the invoking collection those elements that satisfy the condition specified by *predicate*. (Added by JDK 8.)

Table 18-1 The Methods Declared by **Collection**

Method	Description
boolean retainAll(Collection<?> *c*)	Removes all elements from the invoking collection except those in *c*. Returns **true** if the collection changed (i.e., elements were removed). Otherwise, returns **false**.
int size( )	Returns the number of elements held in the invoking collection.
default Spliterator<E> spliterator( )	Returns a spliterator to the invoking collections. (Added by JDK 8.)
default Stream<E> stream( )	Returns a stream that uses the invoking collection as its source for elements. The stream is sequential. (Added by JDK 8.)
Object[ ] toArray( )	Returns an array that contains all the elements stored in the invoking collection. The array elements are copies of the collection elements.
<T> T[ ] toArray(T *array*[ ])	Returns an array that contains the elements of the invoking collection. The array elements are copies of the collection elements. If the size of *array* equals the number of elements, these are returned in *array*. If the size of *array* is less than the number of elements, a new array of the necessary size is allocated and returned. If the size of *array* 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 a subtype of *array*.

Table 18-1 The Methods Declared by **Collection** *(continued)*

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( )**. Beginning with JDK 8, to remove an element only if it statisfies some condition, you can use **removeIf( )**. (**Predicate** is a functional interface added by JDK 8. See Chapter 19.) 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 invoking 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. 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 new **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. (See Chapter 29 for a detailed discussion of the new **Stream** interface.)

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 18-2. 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.

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( )**. (**UnaryOperator** is a functional interface added by JDK 8. See Chapter 19.)

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**.

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

Method	Description
void add(int *index*, E *obj*)	Inserts *obj* into the invoking list at the index passed in *index*. Any preexisting elements at or beyond the point of insertion are shifted up. Thus, no elements are overwritten.
boolean addAll(int *index*, Collection<? extends E> *c*)	Inserts all elements of *c* into the invoking list at the index passed in *index*. Any preexisting elements at or beyond the point of insertion are shifted up. Thus, no elements are overwritten. Returns **true** if the invoking list changes and returns **false** otherwise.
E get(int *index*)	Returns the object stored at the specified index within the invoking collection.
int indexOf(Object *obj*)	Returns the index of the first instance of *obj* in the invoking list. If *obj* is not an element of the list, –1 is returned.
int lastIndexOf(Object *obj*)	Returns the index of the last instance of *obj* in the invoking list. If *obj* is not an element of the list, –1 is returned.
ListIterator<E> listIterator( )	Returns an iterator to the start of the invoking list.
ListIterator<E> listIterator(int *index*)	Returns an iterator to the invoking list that begins at the specified *index*.
E remove(int *index*)	Removes the element at position *index* from the invoking list and returns the deleted element. The resulting list is compacted. That is, the indexes of subsequent elements are decremented by one.
default void replaceAll(UnaryOperator<E> *opToApply*)	Updates each element in the list with the value obtained from the *opToApply* function. (Added by JDK 8.)
E set(int *index*, E *obj*)	Assigns *obj* to the location specified by *index* within the invoking list. Returns the old value.
default void sort(Comparator<? super E> *comp*)	Sorts the list using the comparator specified by *comp*. (Added by JDK 8.)
List<E> subList(int *start*, int *end*)	Returns a list that includes elements from *start* to *end*–1 in the invoking list. Elements in the returned list are also referenced by the invoking object.

Table 18-2 The Methods Declared by **List**

false if an attempt is made to add duplicate elements to a set. 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.

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 18-3. 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.

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

Method	Description
Comparator<? 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 *end* 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 *start* and *end*–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 *start* that are contained in the sorted set. Elements in the returned set are also referenced by the invoking object.

Table 18-3 The Methods Declared by **SortedSet**

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 18-4. A

Method	Description
E ceiling(E *obj*)	Searches the set for the smallest element *e* such that *e* >= *obj*. 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 *obj*)	Searches the set for the largest element *e* such that *e* <= *obj*. If such an element is found, it is returned. Otherwise, **null** is returned.
NavigableSet<E> headSet(E *upperBound*, boolean *incl*)	Returns a **NavigableSet** that includes all elements from the invoking set that are less than *upperBound*. If *incl* is **true**, then an element equal to *upperBound* is included. The resulting set is backed by the invoking set.
E higher(E *obj*)	Searches the set for the largest element *e* such that *e* > *obj*. If such an element is found, it is returned. Otherwise, **null** is returned.
E lower(E *obj*)	Searches the set for the largest element *e* such that *e* < *obj*. 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 *lowerBound*, boolean *lowIncl*, E *upperBound*, boolean *highIncl*)	Returns a **NavigableSet** that includes all elements from the invoking set that are greater than *lowerBound* and less than *upperBound*. If *lowIncl* is **true**, then an element equal to *lowerBound* is included. If *highIncl* is **true**, then an element equal to *upperBound* is included. The resulting set is backed by the invoking set.
NavigableSet<E> tailSet(E *lowerBound*, boolean *incl*)	Returns a **NavigableSet** that includes all elements from the invoking set that are greater than *lowerBound*. If *incl* is **true**, then an element equal to *lowerBound* is included. The resulting set is backed by the invoking set.

Table 18-4 The Methods Declared by **NavigableSet**

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.

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 18-5.

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.

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 *obj* to the queue. Returns **true** if *obj* 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.

Table 18-5 The Methods Declared by **Queue**

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 18-6. Several

Method	Description
void addFirst(E *obj*)	Adds *obj* to the head of the deque. Throws an **IllegalStateException** if a capacity-restricted deque is out of space.
void addLast(E *obj*)	Adds *obj* 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 *obj*)	Attempts to add *obj* to the head of the deque. Returns **true** if *obj* was added and **false** otherwise. Therefore, this method returns **false** when an attempt is made to add *obj* to a full, capacity-restricted deque.
boolean offerLast(E *obj*)	Attempts to add *obj* to the tail of the deque. Returns **true** if *obj* 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.

Table 18-6 The Methods Declared by **Deque**

Method	Description
void push(E *obj*)	Adds *obj* to the head of the deque. Throws an **IllegalStateException** if a capacity-restricted deque is out of space.
E removeFirst( )	Returns the element at the head of the deque, removing the element in the process. It throws **NoSuchElementException** if the deque is empty.
boolean removeFirstOccurrence(Object *obj*)	Removes the first occurrence of *obj* from the deque. Returns **true** if successful and **false** if the deque did not contain *obj*.
E removeLast( )	Returns the element at the tail of the deque, removing the element in the process. It throws **NoSuchElementException** if the deque is empty.
boolean removeLastOccurrence(Object *obj*)	Removes the last occurrence of *obj* from the deque. Returns **true** if successful and **false** if the deque did not contain *obj*.

Table 18-6 The Methods Declared by **Deque** *(continued)*

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.

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

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

The 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.

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

ArrayList has the constructors shown here:

ArrayList( )
ArrayList(Collection<? 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 two versions of **toArray( )**, which are shown again here for your convenience:

object[ ] toArray( )
<T> T[ ] toArray(T *array*[ ])

The first returns an array of **Object**. The second returns an array of elements that have the same type as **T**. Normally, the second form is more convenient because it returns the proper type of array. The following program demonstrates its use:

```
// 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 **Integer**s. 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 capacity*) of the hash set from its arguments. The fill ratio must be between 0.0 and 1.0, and it determines how full the hash set can be before it is resized upward. Specifically, when the number of elements is greater than the capacity of the hash set multiplied by its fill ratio, the hash set is expanded. For constructors that do not take a fill ratio, 0.75 is used.

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

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

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

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

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

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

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

The following is the output from this program:

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

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

The LinkedHashSet Class

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

class LinkedHashSet<E>

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

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

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

which is the order in which the elements were inserted.

The TreeSet Class

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

TreeSet is a generic class that has this declaration:

class TreeSet<E>

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

TreeSet has the following constructors:

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

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

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

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

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

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

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

The output from this program is shown here:

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

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

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

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

The output from this statement is shown here:

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

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

The PriorityQueue Class

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

class PriorityQueue<E>

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

PriorityQueue defines the six constructors shown here:

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

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

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

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

Comparator<? super E> comparator( )

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

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

The ArrayDeque Class

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

class ArrayDeque<E>

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

ArrayDeque defines the following constructors:

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

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

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

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

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

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

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

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

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

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

The output is shown here:

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


The EnumSet Class

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

class EnumSet<E extends Enum<E>>

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

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

Accessing a Collection via an Iterator

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

interface Iterator<E>
interface ListIterator<E>

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

```
    System.out.println();

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

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

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

Output from this program is shown here:

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

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

Arrays

The **Arrays** class provides various methods that are useful when working with arrays. These methods help bridge the gap between collections and arrays. Each method defined by **Arrays** is examined in this section.

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

static <T> List asList(T... *array*)

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

The **binarySearch( )** method uses a binary search to find a specified value. This method must be applied to sorted arrays. Here are some of its forms. (Additional forms let you search a subrange):

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Here, *value* is assigned to all elements in *array*. The second version of the **fill( )** method assigns a value to a subset of an array.

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

static void sort(byte *array*[ ])
static void sort(char *array*[ ])
static void sort(double *array*[ ])
static void sort(float *array*[ ])
static void sort(int *array*[ ])
static void sort(long *array*[ ])

static void sort(short *array*[ ])
static void sort(Object *array*[ ])
static <T> void sort(T *array*[ ], Comparator<? super T> *c*)

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

    // Sort and display the array.
```

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

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

    System.out.println(index);
  }

  static void display(int array[]) {
    for(int i: array)
      System.out.print(i + " ");

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

The following is the output from this program:

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

The Legacy Classes and Interfaces

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

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

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

Dictionary	Hashtable	Properties	Stack	Vector

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