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>
    thenComparingInt(ToIntFunction<? super T> getKey)

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

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

Finally, **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>
    comparingInt(ToIntFunction<? super T> getKey)

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

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

Using a Comparator

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

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

// A reverse comparator for strings.
class MyComp implements Comparator<String> {
  public int compare(String aStr, String bStr) {

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

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

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

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

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

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

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

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

Look closely at the **MyComp** class, which implements **Comparator** by implementing **compare( )**. (As explained earlier, overriding **equals( )** is neither necessary nor common. It is also not necessary to override the default methods.) 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. You might want to try this on your own. Just follow the same general approach described for the **CompDemo2** example shown earlier.

The Collection Algorithms

The Collections Framework defines several algorithms that can be applied to collections and maps. These algorithms are defined as static methods within the **Collections** class. They are summarized in Table 19-15.

Method	Description
static <T> boolean addAll(Collection <? super T> *c*, T... *elements*)	Inserts the elements specified by *elements* into the collection specified by *c*. Returns **true** if the elements were added and **false** otherwise.
static <T> Queue<T> asLifoQueue(Deque<T> *c*)	Returns a last-in, first-out view of *c*.
static <T> int binarySearch(List<? extends T> *list*, T *value*, Comparator<? super T> *c*)	Searches for *value* in *list* ordered according to *c*. Returns the position of value in *list*, or a negative value if *value* is not found.
static <T> int binarySearch(List<? extends Comparable<? super T>> *list*, T *value*)	Searches for *value* in *list*. The list must be sorted. Returns the position of *value* in *list*, or a negative value if *value* is not found.
static <E> Collection<E> checkedCollection(Collection<E> *c*, Class<E> *t*)	Returns a run-time type-safe view of a collection. An attempt to insert an incompatible element will cause a **ClassCastException**.
static <E> List<E> checkedList(List<E> *c*, Class<E> *t*)	Returns a run-time type-safe view of a **List**. An attempt to insert an incompatible element will cause a **ClassCastException**.
static <K, V> Map<K, V> checkedMap(Map<K, V> *c*, Class<K> *keyT*, Class<V> *valueT*)	Returns a run-time type-safe view of a **Map**. An attempt to insert an incompatible element will cause a **ClassCastException**.
static <K, V> NavigableMap<K, V> checkedNavigableMap(NavigableMap<K, V> *nm*, Class<E> *keyT*, Class<V> *valueT*)	Returns a run-time type-safe view of a **NavigableMap**. An attempt to insert an incompatible element will cause a **ClassCastException**.
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 *list2* to *list1*.
static boolean disjoint(Collection<?> *a*, Collection<?> *b*)	Compares the elements in *a* to elements in *b*. Returns **true** if the two collections contain no common elements (i.e., the collections contain disjoint sets of elements). Otherwise, returns **false**.
static <T> Enumeration<T> emptyEnumeration( )	Returns an empty enumeration, which is an enumeration with no elements.
static <T> Iterator<T> emptyIterator( )	Returns an empty iterator, which is an iterator with no elements.
static <T> List<T> emptyList( )	Returns an immutable, empty **List** object of the inferred type.
static <T> ListIterator<T> emptyListIterator( )	Returns an empty list iterator, which is a list iterator that has no elements.
static <K, V> Map<K, V> emptyMap( )	Returns an immutable, empty **Map** object of the inferred type.
static <K, V> NavigableMap<K, V> emptyNavigableMap( )	Returns an immutable, empty **NavigableMap** object of the inferred type.
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 *c*. (See "The Enumeration Interface," later in this chapter.)
static <T> void fill(List<? super T> *list*, T *obj*)	Assigns *obj* to each element of *list*.

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

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

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

Table 19-15 The Algorithms Defined by **Collections**

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), **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**. 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 19-16.

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

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

Table 19-16 The Legacy Methods Defined by **Vector**

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 19-17.

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 *element*)	Pushes *element* onto the stack. *element* is also returned.
int search(Object *element*)	Searches for *element* in the stack. If found, its offset from the top of the stack is returned. Otherwise, –1 is returned.

Table 19-17 The Methods Defined by **Stack**

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 19-18.

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 *key*. If *key* 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 *key* is not already in the dictionary; returns the previous value associated with *key* if *key* is already in the dictionary.
V remove(Object *key*)	Removes *key* and its value. Returns the value associated with *key*. If *key* is not in the dictionary, a **null** is returned.
int size( )	Returns the number of entries in the dictionary.

Table 19-18 The Abstract Methods Defined by **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.

REMEMBER The **Dictionary** class is obsolete. You should implement the **Map** interface to obtain key/value storage functionality.

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 19-19. Several methods throw **NullPointerException** if an attempt is made to use a **null** key or value.

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

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

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 19-20. **Properties** also contains one deprecated method: **save( )**. This was replaced by **store( )** because **save( )** did not handle errors correctly.

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

Table 19-20 The Methods Defined by **Properties**

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. Frankly, the types of problems that will benefit from a collections-based solution are limited only by your imagination. One last point: In Chapter 29, the stream API is discussed. Because streams are integrated with collections, consider using a stream when operating on a collection.

CHAPTER

20

java.util Part 2: More Utility Classes

This chapter continues our discussion of **java.util** by examining those classes and interfaces that are not part of the Collections Framework. These include classes that support timers, work with dates, compute random numbers, and bundle resources. Also covered are the **Formatter** and **Scanner** classes which make it easy to write and read formatted data, and the **Optional** class, which simplifies handling situations in which a value may be absent. Finally, the subpackages of **java.util** are summarized at the end of this chapter. Of particular interest is **java.util.function**, which defines several standard functional interfaces. One last point: the **Observer** interface and the **Observable** class packaged in **java.util**. were deprecated by JDK 9. For this reason they are not discussed here.

StringTokenizer

The processing of text often consists of parsing a formatted input string. *Parsing* is the division of text into a set of discrete parts, or *tokens*, which in a certain sequence can convey a semantic meaning. The **StringTokenizer** class provides the first step in this parsing process, often called the *lexer* (lexical analyzer) or *scanner*. **StringTokenizer** implements the **Enumeration** interface. Therefore, given an input string, you can enumerate the individual tokens contained in it using **StringTokenizer**. Before we begin, it is important to point out that **StringTokenizer** is described here primarily for the benefit of those programmers working with legacy code. For new code, regular expressions, discussed in Chapter 30, offer a more modern alternative.

To use **StringTokenizer**, you specify an input string and a string that contains delimiters. *Delimiters* are characters that separate tokens. Each character in the delimiters string is considered a valid delimiter—for example, **",;:"** sets the delimiters to a comma, semicolon, and colon. The default set of delimiters consists of the whitespace characters: space, tab, form feed, newline, and carriage return.

The **StringTokenizer** constructors are shown here:

StringTokenizer(String *str*)
StringTokenizer(String *str*, String *delimiters*)
StringTokenizer(String *str*, String *delimiters*, boolean *delimAsToken*)

In all versions, *str* is the string that will be tokenized. In the first version, the default delimiters are used. In the second and third versions, *delimiters* is a string that specifies the delimiters. In the third version, if *delimAsToken* is **true**, then the delimiters are also returned as tokens when the string is parsed. Otherwise, the delimiters are not returned. Delimiters are not returned as tokens by the first two forms.

Once you have created a **StringTokenizer** object, the **nextToken( )** method is used to extract consecutive tokens. The **hasMoreTokens( )** method returns **true** while there are more tokens to be extracted. Since **StringTokenizer** implements **Enumeration**, the **hasMoreElements( )** and **nextElement( )** methods are also implemented, and they act the same as **hasMoreTokens( )** and **nextToken( )**, respectively. The **StringTokenizer** methods are shown in Table 20-1.

Method	Description
int countTokens( )	Using the current set of delimiters, the method determines the number of tokens left to be parsed and returns the result.
boolean hasMoreElements( )	Returns **true** if one or more tokens remain in the string and returns **false** if there are none.
boolean hasMoreTokens( )	Returns **true** if one or more tokens remain in the string and returns **false** if there are none.
Object nextElement( )	Returns the next token as an **Object**.
String nextToken( )	Returns the next token as a **String**.
String nextToken(String *delimiters*)	Returns the next token as a **String** and sets the delimiters string to that specified by *delimiters*.

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

Here is an example that creates a **StringTokenizer** to parse "key=value" pairs. Consecutive sets of "key=value" pairs are separated by a semicolon.

```
// Demonstrate StringTokenizer.
import java.util.StringTokenizer;

class STDemo {
  static String in = "title=Java: The Complete Reference;" +
    "author=Schildt;" +
    "publisher=Oracle Press;" +
    "copyright=2019";

  public static void main(String args[]) {
    StringTokenizer st = new StringTokenizer(in, "=;");

    while(st.hasMoreTokens()) {
      String key = st.nextToken();
      String val = st.nextToken();
      System.out.println(key + "\t" + val);
    }
  }
}
```

The output from this program is shown here:

```
title Java: The Complete Reference
author Schildt
publisher Oracle Press
copyright 2019
```

BitSet

A **BitSet** class creates a special type of array that holds bit values in the form of **boolean** values. This array can increase in size as needed. This makes it similar to a vector of bits. The **BitSet** constructors are shown here:

BitSet( )
BitSet(int *size*)

The first version creates a default object. The second version allows you to specify its initial size (that is, the number of bits that it can hold). All bits are initialized to **false**.

BitSet defines the methods listed in Table 20-2.

Method	Description
void and(BitSet *bitSet*)	ANDs the contents of the invoking **BitSet** object with those specified by *bitSet*. The result is placed into the invoking object.
void andNot(BitSet *bitSet*)	For each set bit in *bitSet*, the corresponding bit in the invoking **BitSet** is cleared.
int cardinality( )	Returns the number of set bits in the invoking object.
void clear( )	Zeros all bits.
void clear(int *index*)	Zeros the bit specified by *index*.
void clear(int *startIndex*, int *endIndex*)	Zeros the bits from *startIndex* to *endIndex*–1.
Object clone( )	Duplicates the invoking **BitSet** object.
boolean equals(Object *bitSet*)	Returns **true** if the invoking bit set is equivalent to the one passed in *bitSet*. Otherwise, the method returns **false**.
void flip(int *index*)	Reverses the bit specified by *index*.
void flip(int *startIndex*, int *endIndex*)	Reverses the bits from *startIndex* to *endIndex*–1.
boolean get(int *index*)	Returns the current state of the bit at the specified index.
BitSet get(int *startIndex*, int *endIndex*)	Returns a **BitSet** that consists of the bits from *startIndex* to *endIndex*–1. The invoking object is not changed.
int hashCode( )	Returns the hash code for the invoking object.
boolean intersects(BitSet *bitSet*)	Returns **true** if at least one pair of corresponding bits within the invoking object and *bitSet* are set.

boolean isEmpty( )	Returns **true** if all bits in the invoking object are cleared.
int length( )	Returns the number of bits required to hold the contents of the invoking **BitSet**. This value is determined by the location of the last set bit.
int nextClearBit(int *startIndex*)	Returns the index of the next cleared bit (that is, the next **false** bit), starting from the index specified by *startIndex*.
int nextSetBit(int *startIndex*)	Returns the index of the next set bit (that is, the next **true** bit), starting from the index specified by *startIndex*. If no bit is set, –1 is returned.
void or(BitSet *bitSet*)	ORs the contents of the invoking **BitSet** object with that specified by *bitSet*. The result is placed into the invoking object.
int previousClearBit(int *startIndex*)	Returns the index of the next cleared bit (that is, the next **false** bit) at or prior to the index specified by *startIndex*. If no cleared bit is found, –1 is returned.
int previousSetBit(int *startIndex*)	Returns the index of the next set bit (that is, the next **true** bit) at or prior to the index specified by *startIndex*. If no set bit is found, –1 is returned.
void set(int *index*)	Sets the bit specified by *index*.
void set(int *index*, boolean *v*)	Sets the bit specified by *index* to the value passed in *v*. **true** sets the bit; **false** clears the bit.
void set(int *startIndex*, int *endIndex*)	Sets the bits from *startIndex* to *endIndex*–1.
void set(int *startIndex*, int *endIndex*, boolean *v*)	Sets the bits from *startIndex* to *endIndex*–1 to the value passed in *v*. **true** sets the bits; **false** clears the bits.
int size( )	Returns the number of bits in the invoking **BitSet** object.
IntStream stream( )	Returns a stream that contains the bit positions, from low to high, that have set bits.
byte[ ] toByteArray( )	Returns a **byte** array that contains the invoking **BitSet** object.
long[ ] toLongArray( )	Returns a **long** array that contains the invoking **BitSet** object.
String toString( )	Returns the string equivalent of the invoking **BitSet** object.
static BitSet valueOf(byte[ ] *v*)	Returns a **BitSet** that contains the bits in *v*.
static BitSet valueOf(ByteBuffer *v*)	Returns a **BitSet** that contains the bits in *v*.
static BitSet valueOf(long[ ] *v*)	Returns a **BitSet** that contains the bits in *v*.
static BitSet valueOf(LongBuffer *v*)	Returns a **BitSet** that contains the bits in *v*.
void xor(BitSet *bitSet*)	XORs the contents of the invoking **BitSet** object with that specified by *bitSet*. The result is placed into the invoking object.

Table 20-2 The Methods Defined by **BitSet**

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

```
// BitSet Demonstration.
import java.util.BitSet;

class BitSetDemo {
  public static void main(String args[]) {
    BitSet bits1 = new BitSet(16);
    BitSet bits2 = new BitSet(16);

    // set some bits
    for(int i=0; i<16; i++) {
      if((i%2) == 0) bits1.set(i);
      if((i%5) != 0) bits2.set(i);
    }

    System.out.println("Initial pattern in bits1: ");
    System.out.println(bits1);
    System.out.println("\nInitial pattern in bits2: ");
    System.out.println(bits2);

    // AND bits
    bits2.and(bits1);
    System.out.println("\nbits2 AND bits1: ");
    System.out.println(bits2);

    // OR bits
    bits2.or(bits1);
    System.out.println("\nbits2 OR bits1: ");
    System.out.println(bits2);

    // XOR bits
    bits2.xor(bits1);
    System.out.println("\nbits2 XOR bits1: ");
    System.out.println(bits2);
  }
}
```

The output from this program is shown here. When **toString( )** converts a **BitSet** object to its string equivalent, each set bit is represented by its bit position.

Cleared bits are not shown.

```
Initial pattern in bits1:
{0, 2, 4, 6, 8, 10, 12, 14}

Initial pattern in bits2:
{1, 2, 3, 4, 6, 7, 8, 9, 11, 12, 13, 14}

bits2 AND bits1:
{2, 4, 6, 8, 12, 14}

bits2 OR bits1:
{0, 2, 4, 6, 8, 10, 12, 14}

bits2 XOR bits1:
{}
```

Optional, OptionalDouble, OptionalInt, and OptionalLong

Beginning with JDK 8, the classes called **Optional**, **OptionalDouble**, **OptionalInt**, and **OptionalLong** offer a way to handle situations in which a value may or may not be present. In the past, you would normally use the value **null** to indicate that no value is present. However, this can lead to null pointer exceptions if an attempt is made to dereference a null reference. As a result, frequent checks for a **null** value were necessary to avoid generating an exception. These classes provide a better way to handle such situations. One other point: These classes are value-based; as such they are immutable and various restrictions apply, such as not using instances for synchronization and avoiding any use of reference equality. Consult the Java documentation for the latest details on value-based classes.

The first and most general of these classes is **Optional**. For this reason, it is the primary focus of this discussion. It is shown here:

class Optional<T>

Here, **T** specifies the type of value stored. It is important to understand that an **Optional** instance can either contain a value of type **T** or be empty. In other words, an **Optional** object does not necessarily contain a value. **Optional** does not define any constructors, but it does define several methods that let you work with **Optional** objects. For example, you can determine if a value is present,

obtain the value if it is present, obtain a default value when no value is present, and construct an **Optional** value. The **Optional** methods are shown in Table 20-3.

Method	Description
static <T> Optional<T> empty( )	Returns an object for which **isPresent( )** returns **false**.
boolean equals(Object *optional*)	Returns **true** if the invoking object equals *optional*. Otherwise, returns **false**.
Optional<T> filter(Predicate<? super T> *condition*)	Returns an **Optional** instance that contains the same value as the invoking object if that value satisfies *condition*. Otherwise, an empty object is returned.
U Optional<U> flatMap(Function<? super T, Optional<U>> *mapFunc*)	Applies the mapping function specified by *mapFunc* to the invoking object if that object contains a value and returns the result. Returns an empty object otherwise.
T get( )	Returns the value in the invoking object. However, if no value is present, **NoSuchElementException** is thrown.
int hashCode( )	Returns a hash code for the value in invoking object. Returns 0 if there is no value.
void ifPresent(Consumer<? super T> *func*)	Calls *func* if a value is present in the invoking object, passing the object to *func*. If no value is present, no action occurs.
void ifPresentOrElse(Consumer<? super T> *func*, Runnable *onEmpty*)	Calls *func* if a value is present in the invoking object, passing the object to *func*. If no value is present, *onEmpty* will be executed.

boolean isEmpty( )	Returns **true** if the invoking object does not contain a value. Returns **false** if a value is present. (Added by JDK 11.)
boolean isPresent( )	Returns **true** if the invoking object contains a value. Returns **false** if no value is present.
U Optional<U> map(Function<? super T, ? extends U>> *mapFunc*)	Applies the mapping function specified by *mapFunc* to the invoking object if that object contains a value and returns the result. Returns an empty object otherwise.
static <T> Optional<T> of(T *val*)	Creates an **Optional** instance that contains *val* and returns the result. The value of *val* must not be **null**.
static <T> Optional<T> ofNullable(T *val*)	Creates an **Optional** instance that contains *val* and returns the result. However, if *val* is **null**, then an empty **Optional** instance is returned.
Optional<T> or(Supplier<? extends Optional<? extends T>> *func*)	If no value is present in the invoking object, calls *func* to construct and return an **Optional** instance that contains a value. Otherwise, returns an **Optional** instance that contains the invoking object's value.
T orElse(T *defVal*)	If the invoking object contains a value, the value is returned. Otherwise, the value specified by *defVal* is returned.
T orElseGet(Supplier<? extends T> *getFunc*)	If the invoking object contains a value, the value is returned. Otherwise, the value obtained from *getFunc* is returned.
T orElseThrow( )	Returns the value in the invoking object. However, if no value is present, **NoSuchElementException** is thrown.
<X extends Throwable> T orElseThrow(Supplier<? extends X> *excFunc*) throws X extends Throwable	Returns the value in the invoking object. However, if no value is present, the exception generated by *excFunc* is thrown.
Stream<T> stream( )	Returns a stream that contains the invoking object's value. If no value is present, the stream will contain no values.
String toString( )	Returns a string corresponding to the invoking object.

Table 20-3 The Methods Defined by **Optional**

The best way to understand **Optional** is to work through an example that uses its core methods. At the foundation of **Optional** are **isPresent( )** and **get( )**. You can determine if a value is present by calling **isPresent( )**. If a value is available, it will return **true**. Otherwise, **false** is returned. If a value is present in an **Optional** instance, you can obtain it by calling **get( )**. However, if you call **get( )** on an object that does not contain a value, **NoSuchElementException** is thrown. For this reason, you should always first confirm that a value is present before calling **get( )** on an **Optional** object. Beginning with JDK 10, the parameterless

version of **orElseThrow( )** can be used instead of **get( )**, and its name adds clarity to the operation. However, the examples in this book will use **get( )** so that the code will compile for readers using earlier versions of Java.

Of course, having to call two methods to retrieve a value adds overhead to each access. Fortunately, **Optional** defines methods that combine the check for a value with the retrieval of the value. One such method is **orElse( )**. If the object on which it is called contains a value, the value is returned. Otherwise, a default value is returned.

Optional does not define any constructors. Instead, you will use one of its methods to create an instance. For example, you can create an **Optional** instance with a specified value by using **of( )**. You can create an instance of **Optional** that does not contain a value by using **empty( )**.

The following program demonstrates these methods:

```
// Demonstrate several Optional<T> methods

import java.util.*;

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

    Optional<String> noVal = Optional.empty();

    Optional<String> hasVal = Optional.of("ABCDEFG");

    if(noVal.isPresent()) System.out.println("This won't be displayed");
    else System.out.println("noVal has no value");

    if(hasVal.isPresent()) System.out.println("The string in hasVal is: " +
                                              hasVal.get());

    String defStr = noVal.orElse("Default String");
    System.out.println(defStr);
  }
}
```

The output is shown here:

```
noVal has no value
The string in hasVal is: ABCDEFG
Default String
```

As the output shows, a value can be obtained from an **Optional** object only if

one is present. This basic mechanism enables **Optional** to prevent null pointer exceptions.

The **OptionalDouble**, **OptionalInt**, and **OptionalLong** classes work much like **Optional**, except that they are designed expressly for use on **double**, **int**, and **long** values, respectively. As such, they specify the methods **getAsDouble( )**, **getAsInt( )**, and **getAsLong( )**, respectively, rather than **get( )**. Also, they do not support the **filter( )**, **ofNullable( )**, **map( )**, **flatMap( ),** and **or( )** methods.

Date

The **Date** class encapsulates the current date and time. Before beginning our examination of **Date**, it is important to point out that it has changed substantially from its original version defined by Java 1.0. When Java 1.1 was released, many of the functions carried out by the original **Date** class were moved into the **Calendar** and **DateFormat** classes, and as a result, many of the original 1.0 **Date** methods were deprecated. Since the deprecated 1.0 methods should not be used for new code, they are not described here.

Date supports the following non-deprecated constructors:

Date( )
Date(long *millisec*)

The first constructor initializes the object with the current date and time. The second constructor accepts one argument that equals the number of milliseconds that have elapsed since midnight, January 1, 1970. The non-deprecated methods defined by **Date** are shown in Table 20-4. **Date** also implements the **Comparable** interface.

Method	Description
boolean after(Date *date*)	Returns **true** if the invoking **Date** object contains a date that is later than the one specified by *date*. Otherwise, it returns **false**.
boolean before(Date *date*)	Returns **true** if the invoking **Date** object contains a date that is earlier than the one specified by *date*. Otherwise, it returns **false**.
Object clone( )	Duplicates the invoking **Date** object.
int compareTo(Date *date*)	Compares the value of the invoking object with that of *date*. Returns 0 if the values are equal. Returns a negative value if the invoking object is earlier than *date*. Returns a positive value if the invoking object is later than *date*.
boolean equals(Object *date*)	Returns **true** if the invoking **Date** object contains the same time and date as the one specified by *date*. Otherwise, it returns **false**.
static Date from(Instant t)	Returns a **Date** object corresponding to the **Instant** object passed in *t*.
long getTime( )	Returns the number of milliseconds that have elapsed since January 1, 1970.
int hashCode( )	Returns a hash code for the invoking object.
void setTime(long *time*)	Sets the time and date as specified by *time*, which represents an elapsed time in milliseconds from midnight, January 1, 1970.
Instant toInstant( )	Returns an **Instant** object corresponding to the invoking **Date** object.
String toString( )	Converts the invoking **Date** object into a string and returns the result.

Table 20-4 The Nondeprecated Methods Defined by **Date**

As you can see by examining Table 20-4, the non-deprecated **Date** features do not allow you to obtain the individual components of the date or time. As the following program demonstrates, you can only obtain the date and time in terms of milliseconds, in its default string representation as returned by **toString( )**, or as an **Instant** object. To obtain more-detailed information about the date and time, you will use the **Calendar** class.

```
// Show date and time using only Date methods.
import java.util.Date;

class DateDemo {
  public static void main(String args[]) {
    // Instantiate a Date object
    Date date = new Date();

    // display time and date using toString()
    System.out.println(date);

    // Display number of milliseconds since midnight, January 1, 1970 GMT
    long msec = date.getTime();
    System.out.println("Milliseconds since Jan. 1, 1970 GMT = " + msec);
  }
}
```

Sample output is shown here:

```
Mon Jan 01 10:52:44 CST 2018
Milliseconds since Jan. 1, 1970 GMT = 1514825564360
```

Calendar

The abstract **Calendar** class provides a set of methods that allows you to convert a time in milliseconds to a number of useful components. Some examples of the type of information that can be provided are year, month, day, hour, minute, and second. It is intended that subclasses of **Calendar** will provide the specific functionality to interpret time information according to their own rules. This is one aspect of the Java class library that enables you to write programs that can operate in international environments. An example of such a subclass is **GregorianCalendar**.

NOTE JDK 8 defined another date and time API in **java.time**, which new applications may want to employ. See Chapter 30.

Calendar provides no public constructors. **Calendar** defines several protected instance variables. **areFieldsSet** is a **boolean** that indicates if the time components have been set. **fields** is an array of **int**s that holds the components of the time. **isSet** is a **boolean** array that indicates if a specific time component has been set. **time** is a **long** that holds the current time for this object. **isTimeSet** is a **boolean** that indicates if the current time has been set.

A sampling of methods defined by **Calendar** are shown in Table 20-5.

Method	Description
abstract void add(int *which*, int *val*)	Adds *val* to the time or date component specified by *which*. To subtract, add a negative value. *which* must be one of the fields defined by **Calendar**, such as **Calendar.HOUR**.
boolean after(Object *calendarObj*)	Returns **true** if the invoking **Calendar** object contains a date that is later than the one specified by *calendarObj*. Otherwise, it returns **false**.
boolean before(Object *calendarObj*)	Returns **true** if the invoking **Calendar** object contains a date that is earlier than the one specified by *calendarObj*. Otherwise, it returns **false**.
final void clear( )	Zeros all time components in the invoking object.
final void clear(int *which*)	Zeros the time component specified by *which* in the invoking object.
Object clone( )	Returns a duplicate of the invoking object.
boolean equals(Object *calendarObj*)	Returns **true** if the invoking **Calendar** object contains a date that is equal to the one specified by *calendarObj*. Otherwise, it returns **false**.
int get(int *calendarField*)	Returns the value of one component of the invoking object. The component is indicated by *calendarField*. Examples of the components that can be requested are **Calendar.YEAR**, **Calendar.MONTH**, **Calendar.MINUTE**, and so forth.
static Locale[ ] getAvailableLocales( )	Returns an array of **Locale** objects that contains the locales for which calendars are available.
static Calendar getInstance( )	Returns a **Calendar** object for the default locale and time zone.
static Calendar getInstance(TimeZone *tz*)	Returns a **Calendar** object for the time zone specified by *tz*. The default locale is used.

static Calendar getInstance(Locale *locale*)	Returns a **Calendar** object for the locale specified by *locale*. The default time zone is used.
static Calendar getInstance(TimeZone *tz*, Locale *locale*)	Returns a **Calendar** object for the time zone specified by *tz* and the locale specified by *locale*.
final Date getTime( )	Returns a **Date** object equivalent to the time of the invoking object.
TimeZone getTimeZone( )	Returns the time zone for the invoking object.
final boolean isSet(int *which*)	Returns **true** if the specified time component is set. Otherwise, it returns **false**.
void set(int *which*, int *val*)	Sets the date or time component specified by *which* to the value specified by *val* in the invoking object. *which* must be one of the fields defined by **Calendar**, such as **Calendar.HOUR**.
final void set(int *year*, int *month*, int *dayOfMonth*)	Sets various date and time components of the invoking object.
final void set(int *year*, int *month*, int *dayOfMonth*, int *hours*, int *minutes*)	Sets various date and time components of the invoking object.
final void set(int *year*, int *month*, int *dayOfMonth*, int *hours*, int *minutes*, int *seconds*)	Sets various date and time components of the invoking object.
final void setTime(Date *d*)	Sets various date and time components of the invoking object. This information is obtained from the **Date** object *d*.
void setTimeZone(TimeZone *tz*)	Sets the time zone for the invoking object to that specified by *tz*.
final Instant toInstant( )	Returns an **Instant** object corresponding to the invoking **Calendar** instance.

Table 20-5 A Sampling of the Methods Defined by **Calendar**

Calendar defines the following **int** constants, which are used when you get or set components of the calendar.

ALL_STYLES	HOUR_OF_DAY	PM
AM	JANUARY	SATURDAY
AM_PM	JULY	SECOND
APRIL	JUNE	SEPTEMBER
AUGUST	LONG	SHORT
DATE	LONG_FORMAT	SHORT_FORMAT
DAY_OF_MONTH	LONG_STANDALONE	SHORT_STANDALONE
DAY_OF_WEEK	MARCH	SUNDAY
DAY_OF_WEEK_IN_MONTH	MAY	THURSDAY
DAY_OF_YEAR	MILLISECOND	TUESDAY
DECEMBER	MINUTE	UNDECIMBER
DST_OFFSET	MONDAY	WEDNESDAY
ERA	MONTH	WEEK_OF_MONTH
FEBRUARY	NARROW_FORMAT	WEEK_OF_YEAR
FIELD_COUNT	NARROW_STANDALONE	YEAR
FRIDAY	NOVEMBER	ZONE_OFFSET
HOUR	OCTOBER	

The following program demonstrates several **Calendar** methods:

```
// Demonstrate Calendar
import java.util.Calendar;

class CalendarDemo {
  public static void main(String args[]) {
    String months[] = {
              "Jan", "Feb", "Mar", "Apr",
              "May", "Jun", "Jul", "Aug",
              "Sep", "Oct", "Nov", "Dec"};

    // Create a calendar initialized with the
    // current date and time in the default
    // locale and timezone.
    Calendar calendar = Calendar.getInstance();

    // Display current time and date information.
    System.out.print("Date: ");
    System.out.print(months[calendar.get(Calendar.MONTH)]);
    System.out.print(" " + calendar.get(Calendar.DATE) + " ");
    System.out.println(calendar.get(Calendar.YEAR));

    System.out.print("Time: ");
    System.out.print(calendar.get(Calendar.HOUR) + ":");
    System.out.print(calendar.get(Calendar.MINUTE) + ":");
    System.out.println(calendar.get(Calendar.SECOND));

    // Set the time and date information and display it.
    calendar.set(Calendar.HOUR, 10);
    calendar.set(Calendar.MINUTE, 29);
    calendar.set(Calendar.SECOND, 22);
    System.out.print("Updated time: ");
    System.out.print(calendar.get(Calendar.HOUR) + ":");
    System.out.print(calendar.get(Calendar.MINUTE) + ":");
    System.out.println(calendar.get(Calendar.SECOND));
  }
}
```

Sample output is shown here:

```
Date: Jan 1 2018
Time: 11:29:39
Updated time: 10:29:22
```

GregorianCalendar

GregorianCalendar is a concrete implementation of a **Calendar** that implements the normal Gregorian calendar with which you are familiar. The **getInstance( )** method of **Calendar** will typically return a **GregorianCalendar** initialized with the current date and time in the default locale and time zone.

GregorianCalendar defines two fields: **AD** and **BC**. These represent the two eras defined by the Gregorian calendar.

There are also several constructors for **GregorianCalendar** objects. The default, **GregorianCalendar( )**, initializes the object with the current date and time in the default locale and time zone. Three more constructors offer increasing levels of specificity:

GregorianCalendar(int *year*, int *month*, int *dayOfMonth*)
GregorianCalendar(int *year*, int *month*, int *dayOfMonth*, int *hours*, int *minutes*)
GregorianCalendar(int *year*, int *month*, int *dayOfMonth*, int *hours*, int *minutes*, int *seconds*)

All three versions set the day, month, and year. Here, *year* specifies the year. The month is specified by *month*, with zero indicating January. The day of the month is specified by *dayOfMonth*. The first version sets the time to midnight. The second version also sets the hours and the minutes. The third version adds seconds.

You can also construct a **GregorianCalendar** object by specifying the locale and/or time zone. The following constructors create objects initialized with the current date and time using the specified time zone and/or locale:

GregorianCalendar(Locale *locale*)
GregorianCalendar(TimeZone *timeZone*)
GregorianCalendar(TimeZone *timeZone*, Locale *locale*)

GregorianCalendar provides an implementation of all the abstract methods in **Calendar**. It also provides some additional methods. Perhaps the most interesting is **isLeapYear( )**, which tests if the year is a leap year. Its form is

boolean isLeapYear(int *year*)

This method returns **true** if *year* is a leap year and **false** otherwise. Two other methods of interest are **from( )** and **toZonedDateTime( )**, which support the

date and time API added by JDK 8 and packaged in **java.time**.

The following program demonstrates **GregorianCalendar**:

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

class GregorianCalendarDemo {
  public static void main(String args[]) {
    String months[] = {
              "Jan", "Feb", "Mar", "Apr",
              "May", "Jun", "Jul", "Aug",
              "Sep", "Oct", "Nov", "Dec"};
    int year;

    // Create a Gregorian calendar initialized
    // with the current date and time in the
    // default locale and timezone.
    GregorianCalendar gcalendar = new GregorianCalendar();

    // Display current time and date information.
    System.out.print("Date: ");
    System.out.print(months[gcalendar.get(Calendar.MONTH)]);
    System.out.print(" " + gcalendar.get(Calendar.DATE) + " ");
    System.out.println(year = gcalendar.get(Calendar.YEAR));

    System.out.print("Time: ");
    System.out.print(gcalendar.get(Calendar.HOUR) + ":");
    System.out.print(gcalendar.get(Calendar.MINUTE) + ":");
    System.out.println(gcalendar.get(Calendar.SECOND));

    // Test if the current year is a leap year
    if(gcalendar.isLeapYear(year)) {
      System.out.println("The current year is a leap year");
    }

    else {
      System.out.println("The current year is not a leap year");
    }
  }
}
```

Sample output is shown here:

```
Date: Jan 1 2018
Time: 1:45:5
The current year is not a leap year
```

TimeZone

Another time-related class is **TimeZone**. The abstract **TimeZone** class allows you to work with time zone offsets from Greenwich mean time (GMT), also referred to as Coordinated Universal Time (UTC). It also computes daylight saving time. **TimeZone** only supplies the default constructor.

A sampling of methods defined by **TimeZone** is given in Table 20-6.

Method	Description
Object clone( )	Returns a **TimeZone**-specific version of **clone( )**.
static String[ ] getAvailableIDs( )	Returns an array of **String** objects representing the names of all time zones.
static String[ ] getAvailableIDs(int *timeDelta*)	Returns an array of **String** objects representing the names of all time zones that are *timeDelta* offset from GMT.
static TimeZone getDefault( )	Returns a **TimeZone** object that represents the default time zone used on the host computer.
String getID( )	Returns the name of the invoking **TimeZone** object.
abstract int getOffset(int *era*, int *year*, int *month*, int *dayOfMonth*, int *dayOfWeek*, int *millisec*)	Returns the offset that should be added to GMT to compute local time. This value is adjusted for daylight saving time. The parameters to the method represent date and time components.
abstract int getRawOffset( )	Returns the raw offset (in milliseconds) that should be added to GMT to compute local time. This value is not adjusted for daylight saving time.
static TimeZone getTimeZone(String *tzName*)	Returns the **TimeZone** object for the time zone named *tzName*.
abstract boolean inDaylightTime(Date *d*)	Returns **true** if the date represented by *d* is in daylight saving time in the invoking object. Otherwise, it returns **false**.
static void setDefault(TimeZone *tz*)	Sets the default time zone to be used on this host. *tz* is a reference to the **TimeZone** object to be used.
void setID(String *tzName*)	Sets the name of the time zone (that is, its ID) to that specified by *tzName*.
abstract void setRawOffset(int *millis*)	Sets the offset in milliseconds from GMT.
ZoneId toZoneId( )	Converts the invoking object into a **ZoneId** and returns the result. **ZoneId** is packaged in **java.time**.
abstract boolean useDaylightTime( )	Returns **true** if the invoking object uses daylight saving time. Otherwise, it returns **false**.

Table 20-6 A Sampling of the Methods Defined by **TimeZone**

SimpleTimeZone

The **SimpleTimeZone** class is a convenient subclass of **TimeZone**. It implements **TimeZone**'s abstract methods and allows you to work with time zones for a Gregorian calendar. It also computes daylight saving time.

SimpleTimeZone defines four constructors. One is

SimpleTimeZone(int *timeDelta*, String *tzName*)

This constructor creates a **SimpleTimeZone** object. The offset relative to Greenwich mean time (GMT) is *timeDelta*. The time zone is named *tzName*.

The second **SimpleTimeZone** constructor is

SimpleTimeZone(int *timeDelta*, String *tzId*, int *dstMonth0*,
int *dstDayInMonth0*, int *dstDay0*, int *time0*,
int *dstMonth1*, int *dstDayInMonth1*, int *dstDay1*,
int *time1*)

Here, the offset relative to GMT is specified in *timeDelta*. The time zone name is passed in *tzId*. The start of daylight saving time is indicated by the parameters *dstMonth0*, *dstDayInMonth0*, *dstDay0*, and *time0*. The end of daylight saving time is indicated by the parameters *dstMonth1*, *dstDayInMonth1*, *dstDay1*, and *time1*.

The third **SimpleTimeZone** constructor is

SimpleTimeZone(int *timeDelta*, String *tzId*, int *dstMonth0*,
int *dstDayInMonth0*, int *dstDay0*, int *time0*,
int *dstMonth1*, int *dstDayInMonth1*,
int *dstDay1*, int *time1*, int *dstDelta*)

Here, *dstDelta* is the number of milliseconds saved during daylight saving time.

The fourth **SimpleTimeZone** constructor is:

SimpleTimeZone(int *timeDelta*, String *tzId*, int *dstMonth0*,
int *dstDayInMonth0*, int *dstDay0*, int *time0*,
int *time0mode*, int *dstMonth1*, int *dstDayInMonth1*,
int *dstDay1*, int *time1*, int *time1mode*, int *dstDelta*)

Here, *time0mode* specifies the mode of the starting time, and *time1mode* specifies the mode of the ending time. Valid mode values include:

STANDARD_TIME	WALL_TIME	UTC_TIME

The time mode indicates how the time values are interpreted. The default mode used by the other constructors is **WALL_TIME**.

Locale

The **Locale** class is instantiated to produce objects that describe a geographical or cultural region. It is one of several classes that provide you with the ability to write programs that can execute in different international environments. For example, the formats used to display dates, times, and numbers are different in various regions.

Internationalization is a large topic that is beyond the scope of this book. However, many programs will only need to deal with its basics, which include setting the current locale.

The **Locale** class defines the following constants that are useful for dealing with several common locales:

CANADA	GERMAN	KOREAN
CANADA_FRENCH	GERMANY	PRC
CHINA	ITALIAN	SIMPLIFIED_CHINESE
CHINESE	ITALY	TAIWAN
ENGLISH	JAPAN	TRADITIONAL_CHINESE
FRANCE	JAPANESE	UK
FRENCH	KOREA	US

For example, the expression **Locale**.**CANADA** represents the **Locale** object for Canada.

The constructors for **Locale** are

Locale(String *language*)
Locale(String *language*, String *country*)
Locale(String *language*, String *country*, String *variant*)

These constructors build a **Locale** object to represent a specific *language* and in the case of the last two, *country*. These values must contain standard language and country codes. Auxiliary variant information can be provided in *variant*.

Locale defines several methods. One of the most important is **setDefault( )**, shown here:

static void setDefault(Locale *localeObj*)

This sets the default locale used by the JVM to that specified by *localeObj*.

Some other interesting methods are the following:

final String getDisplayCountry( )

final String getDisplayLanguage( )
final String getDisplayName( )

These return human-readable strings that can be used to display the name of the country, the name of the language, and the complete description of the locale.

The default locale can be obtained using **getDefault( )**, shown here:

static Locale getDefault( )

JDK 7 added significant upgrades to the **Locale** class that support Internet Engineering Task Force (IETF) BCP 47, which defines tags for identifying languages, and Unicode Technical Standard (UTS) 35, which defines the Locale Data Markup Language (LDML). Support for BCP 47 and UTS 35 caused several features to be added to **Locale**, including several new methods and the **Locale.Builder** class. Among others, new methods include **getScript( )**, which obtains the locale's script, and **toLanguageTag( )**, which obtains a string that contains the locale's language tag. The **Locale.Builder** class constructs **Locale** instances. It ensures that a locale specification is well-formed as defined by BCP 47. (The **Locale** constructors do not provide such a check.) Several new methods were also added to **Locale** by JDK 8. Among these are methods that support filtering, extensions, and lookups. JDK 9 added a method called **getISOCountries( )**, which returns a collection of country codes for a given **Locale.IsoCountryCode** enumeration value.

Calendar and **GregorianCalendar** are examples of classes that operate in a locale-sensitive manner. **DateFormat** and **SimpleDateFormat** also depend on the locale.

Random

The **Random** class is a generator of pseudorandom numbers. These are called *pseudorandom* numbers because they are simply uniformly distributed sequences. **Random** defines the following constructors:

Random( )
Random(long *seed*)

The first version creates a number generator that uses a reasonably unique seed. The second form allows you to specify a seed value manually.

If you initialize a **Random** object with a seed, you define the starting point for the random sequence. If you use the same seed to initialize another **Random**

object, you will extract the same random sequence. If you want to generate different sequences, specify different seed values. One way to do this is to use the current time to seed a **Random** object. This approach reduces the possibility of getting repeated sequences.

The core public methods defined by **Random** are shown in Table 20-7. These are the methods that have been available in **Random** for several years (many since Java 1.0) and are widely used.

Method	Description
boolean nextBoolean( )	Returns the next **boolean** random number.
void nextBytes(byte *vals*[ ])	Fills *vals* with randomly generated values.
double nextDouble( )	Returns the next **double** random number.
float nextFloat( )	Returns the next **float** random number.
double nextGaussian( )	Returns the next Gaussian random number.
int nextInt( )	Returns the next **int** random number.
int nextInt(int *n*)	Returns the next **int** random number within the range zero to *n*.
long nextLong( )	Returns the next **long** random number.
void setSeed(long *newSeed*)	Sets the seed value (that is, the starting point for the random number generator) to that specified by *newSeed*.

Table 20-7 The Core Methods Defined by **Random**

As you can see, there are seven types of random numbers that you can extract from a **Random** object. Random Boolean values are available from **nextBoolean( )**. Random bytes can be obtained by calling **nextBytes( )**. Integers can be extracted via the **nextInt( )** method. Long integers can be obtained with **nextLong( )**. The **nextFloat( )** and **nextDouble( )** methods return **float** and **double** values, respectively, between 0.0 and 1.0. Finally, **nextGaussian( )** returns a **double** value centered at 0.0 with a standard deviation of 1.0. This is what is known as a *bell curve*.

Here is an example that demonstrates the sequence produced by **nextGaussian( )**. It obtains 100 random Gaussian values and averages these values. The program also counts the number of values that fall within two standard deviations, plus or minus, using increments of 0.5 for each category. The result is graphically displayed sideways on the screen.

```
// Demonstrate random Gaussian values.
import java.util.Random;
class RandDemo {
  public static void main(String args[]) {
    Random r = new Random();
    double val;
    double sum = 0;
    int bell[] = new int[10];

    for(int i=0; i<100; i++) {
      val = r.nextGaussian();
      sum += val;
      double t = -2;

      for(int x=0; x<10; x++, t += 0.5)
        if(val < t) {
          bell[x]++;
          break;
        }
    }
    System.out.println("Average of values: " +
                       (sum/100));

    // display bell curve, sideways
    for(int i=0; i<10; i++) {
      for(int x=bell[i]; x>0; x--)
        System.out.print("*");
      System.out.println();
    }
  }
}
```

Here is a sample program run. As you can see, a bell-like distribution of numbers is obtained.

```
Average of values: 0.0702235271133344
**
*******
******
***************
******************
*****************
*************
**********
```

```
********
***
```

JDK 8 added three methods to **Random** that support the stream API (see Chapter 29). They are called **doubles( )**, **ints( )**, and **longs( )**, and each returns a reference to a stream that contains a sequence of pseudorandom values of the specified type. Each method defines several overloads. Here are their simplest forms:

```
DoubleStream doubles( )

IntStream ints( )

LongStream longs( )
```

The **doubles( )** method returns a stream that contains pseudorandom **double** values. (The range of these values will be less than 1.0 but greater than or equal to 0.0.) The **ints( )** method returns a stream that contains pseudorandom **int** values. The **longs( )** method returns a stream that contains pseudorandom **long** values. For these three methods, the stream returned is effectively infinite. Several overloads of each method are provided that let you specify the size of the stream, an origin, and an upper bound.

Timer and TimerTask

An interesting and useful feature offered by **java.util** is the ability to schedule a task for execution at some future time. The classes that support this are **Timer** and **TimerTask**. Using these classes, you can create a thread that runs in the background, waiting for a specific time. When the time arrives, the task linked to that thread is executed. Various options allow you to schedule a task for repeated execution, and to schedule a task to run on a specific date. Although it was always possible to manually create a task that would be executed at a specific time using the **Thread** class, **Timer** and **TimerTask** greatly simplify this process.

Timer and **TimerTask** work together. **Timer** is the class that you will use to schedule a task for execution. The task being scheduled must be an instance of **TimerTask**. Thus, to schedule a task, you will first create a **TimerTask** object and then schedule it for execution using an instance of **Timer**.

TimerTask implements the **Runnable** interface; thus, it can be used to create a thread of execution. Its constructor is shown here:

protected TimerTask( )

TimerTask defines the methods shown in Table 20-8. Notice that **run( )** is abstract, which means that it must be overridden. The **run( )** method, defined by the **Runnable** interface, contains the code that will be executed. Thus, the easiest way to create a timer task is to extend **TimerTask** and override **run( )**.

Method	Description
boolean cancel( )	Terminates the task. Returns **true** if an execution of the task is prevented. Otherwise, returns **false**.
abstract void run( )	Contains the code for the timer task.
long scheduledExecutionTime( )	Returns the time at which the last execution of the task was scheduled to have occurred.

Table 20-8 The Methods Defined by **TimerTask**

Once a task has been created, it is scheduled for execution by an object of type **Timer**. The constructors for **Timer** are shown here:

Timer( )
Timer(boolean *DThread*)
Timer(String *tName*)
Timer(String *tName*, boolean *DThread*)

The first version creates a **Timer** object that runs as a normal thread. The second uses a daemon thread if *DThread* is **true**. A daemon thread will execute only as long as the rest of the program continues to execute. The third and fourth constructors allow you to specify a name for the **Timer** thread. The methods defined by **Timer** are shown in Table 20-9.

Method	Description
void cancel( )	Cancels the timer thread.
int purge( )	Deletes canceled tasks from the timer's queue.
void schedule(TimerTask *TTask*, long *wait*)	*TTask* is scheduled for execution after the period passed in *wait* has elapsed. The *wait* parameter is specified in milliseconds.
void schedule(TimerTask *TTask*, long *wait*, long *repeat*)	*TTask* is scheduled for execution after the period passed in *wait* has elapsed. The task is then executed repeatedly at the interval specified by *repeat*. Both *wait* and *repeat* are specified in milliseconds.
void schedule(TimerTask *TTask*, Date *targetTime*)	*TTask* is scheduled for execution at the time specified by *targetTime*.
void schedule(TimerTask *TTask*, Date *targetTime*, long *repeat*)	*TTask* is scheduled for execution at the time specified by *targetTime*. The task is then executed repeatedly at the interval passed in *repeat*. The *repeat* parameter is specified in milliseconds.
void scheduleAtFixedRate(TimerTask *TTask*, long *wait*, long *repeat*)	*TTask* is scheduled for execution after the period passed in *wait* has elapsed. The task is then executed repeatedly at the interval specified by *repeat*. Both *wait* and *repeat* are specified in milliseconds. The time of each repetition is relative to the first execution, not the preceding execution. Thus, the overall rate of execution is fixed.
void scheduleAtFixedRate(TimerTask *TTask*, Date *targetTime*, long *repeat*)	*TTask* is scheduled for execution at the time specified by *targetTime*. The task is then executed repeatedly at the interval passed in *repeat*. The *repeat* parameter is specified in milliseconds. The time of each repetition is relative to the first execution, not the preceding execution. Thus, the overall rate of execution is fixed.

Table 20-9 The Methods Defined by **Timer**

Once a **Timer** has been created, you will schedule a task by calling **schedule( )** on the **Timer** that you created. As Table 20-9 shows, there are several forms of **schedule( )** which allow you to schedule tasks in a variety of ways.

If you create a non-daemon task, then you will want to call **cancel( )** to end the task when your program ends. If you don't do this, then your program may "hang" for a period of time.

The following program demonstrates **Timer** and **TimerTask**. It defines a timer task whose **run( )** method displays the message "Timer task executed." This task is scheduled to run once every half second after an initial delay of one second.

```
// Demonstrate Timer and TimerTask.

import java.util.*;

class MyTimerTask extends TimerTask {
  public void run() {
    System.out.println("Timer task executed.");
  }
}

class TTest {
  public static void main(String args[]) {
    MyTimerTask myTask = new MyTimerTask();
    Timer myTimer = new Timer();

    /* Set an initial delay of 1 second,
       then repeat every half second.
    */
    myTimer.schedule(myTask, 1000, 500);

    try {
      Thread.sleep(5000);
    } catch (InterruptedException exc) {}

    myTimer.cancel();
  }
}
```

Currency

The **Currency** class encapsulates information about a currency. It defines no constructors. The methods supported by **Currency** are shown in Table 20-10. The following program demonstrates **Currency**:

Method	Description
static Set<Currency> getAvailableCurrencies( )	Returns a set of the supported currencies.
String getCurrencyCode( )	Returns the code (as defined by ISO 4217) that describes the invoking currency.
int getDefaultFractionDigits( )	Returns the number of digits after the decimal point that are normally used by the invoking currency. For example, there are two fractional digits normally used for dollars.
String getDisplayName( )	Returns the name of the invoking currency for the default locale.
String getDisplayName(Locale *loc*)	Returns the name of the invoking currency for the specified locale.
static Currency getInstance(Locale *localeObj*)	Returns a **Currency** object for the locale specified by *localeObj*.
static Currency getInstance(String *code*)	Returns a **Currency** object associated with the currency code passed in *code*.
int getNumericCode( )	Returns the numeric code (as defined by ISO 4217) for the invoking currency.
String getNumericCodeAsString( )	Returns in string form the numeric code (as defined by ISO 4217) for the invoking currency.
String getSymbol( )	Returns the currency symbol (such as $) for the invoking object.
String getSymbol(Locale *localeObj*)	Returns the currency symbol (such as $) for the locale passed in *localeObj*.
String toString( )	Returns the currency code for the invoking object.

Table 20-10 The Methods Defined by **Currency**

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

class CurDemo {
  public static void main(String args[]) {
    Currency c;

    c = Currency.getInstance(Locale.US);

    System.out.println("Symbol: " + c.getSymbol());
    System.out.println("Default fractional digits: " +
                       c.getDefaultFractionDigits());
  }
}
```

The output is shown here:

```
Symbol: $
Default fractional digits: 2
```

Formatter

At the core of Java's support for creating formatted output is the **Formatter** class. It provides *format conversions* that let you display numbers, strings, and time and date in virtually any format you like. It operates in a manner similar to the C/C++ **printf( )** function, which means that if you are familiar with C/C++, then learning to use **Formatter** will be very easy. It also further streamlines the conversion of C/C++ code to Java. If you are not familiar with C/C++, it is still quite easy to format data.

NOTE Although Java's **Formatter** class operates in a manner very similar to the C/C++ **printf( )** function, there are some differences, and some new features. Therefore, if you have a C/C++ background, a careful reading is advised.

The Formatter Constructors

Before you can use **Formatter** to format output, you must create a **Formatter** object. In general, **Formatter** works by converting the binary form of data used by a program into formatted text. It stores the formatted text in a buffer, the contents of which can be obtained by your program whenever they are needed. It is possible to let **Formatter** supply this buffer automatically, or you can specify the buffer explicitly when a **Formatter** object is created. It is also possible to have **Formatter** output its buffer to a file.

The **Formatter** class defines many constructors, which enable you to construct a **Formatter** in a variety of ways. Here is a sampling:

Formatter( )

Formatter(Appendable *buf*)

Formatter(Appendable *buf*, Locale *loc*)

Formatter(String *filename*)
throws FileNotFoundException

Formatter(String *filename*, String *charset*)
throws FileNotFoundException, UnsupportedEncodingException

Formatter(File *outF*)
throws FileNotFoundException

Formatter(OutputStream *outStrm*)

Here, *buf* specifies a buffer for the formatted output. If *buf* is null, then **Formatter** automatically allocates a **StringBuilder** to hold the formatted output. The *loc* parameter specifies a locale. If no locale is specified, the default locale is used. The *filename* parameter specifies the name of a file that will receive the formatted output. The *charset* parameter specifies the character set. If no character set is specified, then the default character set is used. The *outF* parameter specifies a reference to an open file that will receive output. The *outStrm* parameter specifies a reference to an output stream that will receive output. When using a file, output is also written to the file.

Perhaps the most widely used constructor is the first, which has no parameters. It automatically uses the default locale and allocates a **StringBuilder** to hold the formatted output.

The Formatter Methods

Formatter defines the methods shown in Table 20-11.

Method	Description
void close( )	Closes the invoking **Formatter**. This causes any resources used by the object to be released. After a **Formatter** has been closed, it cannot be reused. An attempt to use a closed **Formatter** results in a **FormatterClosedException**.
void flush( )	Flushes the format buffer. This causes any output currently in the buffer to be written to the destination. This applies mostly to a **Formatter** tied to a file.
Formatter format(String *fmtString*, Object ... *args*)	Formats the arguments passed via *args* according to the format specifiers contained in *fmtString*. Returns the invoking object.
Formatter format(Locale *loc*, String *fmtString*, Object ... *args*)	Formats the arguments passed via *args* according to the format specifiers contained in *fmtString*. The locale specified by *loc* is used for this format. Returns the invoking object.
IOException ioException( )	If the underlying object that is the destination for output throws an **IOException**, then this exception is returned. Otherwise, **null** is returned.
Locale locale( )	Returns the invoking object's locale.
Appendable out( )	Returns a reference to the underlying object that is the destination for output.
String toString( )	Returns a **String** containing the formatted output.

Table 20-11 The Methods Defined by **Formatter**

Formatting Basics

After you have created a **Formatter**, you can use it to create a formatted string. To do so, use the **format( )** method. The version we will use is shown here:

Formatter format(String *fmtString*, Object ... *args*)

The *fmtSring* consists of two types of items. The first type is composed of characters that are simply copied to the output buffer. The second type contains *format specifiers* that define the way the subsequent arguments are displayed.

In its simplest form, a format specifier begins with a percent sign followed by the format *conversion specifier*. All format conversion specifiers consist of a single character. For example, the format specifier for floating-point data is **%f**. In general, there must be the same number of arguments as there are format specifiers, and the format specifiers and the arguments are matched in order from left to right. For example, consider this fragment:

```
Formatter fmt = new Formatter();
fmt.format("Formatting %s is easy %d %f", "with Java", 10, 98.6);
```

This sequence creates a **Formatter** that contains the following string:

```
Formatting with Java is easy 10 98.600000
```

In this example, the format specifiers, **%s**, **%d**, and **%f**, are replaced with the arguments that follow the format string. Thus, **%s** is replaced by "with Java", **%d** is replaced by 10, and **%f** is replaced by 98.6. All other characters are simply used as-is. As you might guess, the format specifier **%s** specifies a string, and **%d** specifies an integer value. As mentioned earlier, the **%f** specifies a floating-point value.

The **format( )** method accepts a wide variety of format specifiers, which are shown in Table 20-12. Notice that many specifiers have both upper- and lowercase forms. When an uppercase specifier is used, then letters are shown in uppercase. Otherwise, the upper- and lowercase specifiers perform the same conversion. It is important to understand that Java type-checks each format specifier against its corresponding argument. If the argument doesn't match, an **IllegalFormatException** is thrown.

Format Specifier	Conversion Applied
%a %A	Floating-point hexadecimal
%b %B	Boolean
%c %C	Character
%d	Decimal integer
%h %H	Hash code of the argument
%e %E	Scientific notation
%f	Decimal floating-point
%g %G	Uses **%e** or **%f**, based on the value being formatted and the precision
%o	Octal integer
%n	Inserts a newline character
%s %S	String
%t %T	Time and date
%x %X	Integer hexadecimal
%%	Inserts a % sign

Table 20-12 The Format Specifiers

Once you have formatted a string, you can obtain it by calling **toString( )**. For example, continuing with the preceding example, the following statement obtains the formatted string contained in **fmt**:

```
String str = fmt.toString();
```

Of course, if you simply want to display the formatted string, there is no reason to first assign it to a **String** object. When a **Formatter** object is passed to **println( )**, for example, its **toString( )** method is automatically called.

Here is a short program that puts together all of the pieces, showing how to

create and display a formatted string:

```
// A very simple example that uses Formatter.
import java.util.*;

class FormatDemo {
  public static void main(String args[]) {
    Formatter fmt = new Formatter();

    fmt.format("Formatting %s is easy %d %f", "with Java", 10, 98.6);

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

One other point: You can obtain a reference to the underlying output buffer by calling **out( )**. It returns a reference to an **Appendable** object.

Now that you know the general mechanism used to create a formatted string, the remainder of this section discusses in detail each conversion. It also describes various options, such as justification, minimum field width, and precision.

Formatting Strings and Characters

To format an individual character, use **%c**. This causes the matching character argument to be output, unmodified. To format a string, use **%s**.

Formatting Numbers

To format an integer in decimal format, use **%d**. To format a floating-point value in decimal format, use **%f**. To format a floating-point value in scientific notation, use **%e**. Numbers represented in scientific notation take this general form:

*x.ddddd*e+/–yy

The **%g** format specifier causes **Formatter** to use either **%f** or **%e**, based on the value being formatted and the precision, which is 6 by default. The following program demonstrates the effect of the **%f** and **%e** format specifiers:

```
// Demonstrate the %f and %e format specifiers.
import java.util.*;

class FormatDemo2 {
  public static void main(String args[]) {
    Formatter fmt = new Formatter();

    for(double i=1.23; i < 1.0e+6; i *= 100) {
      fmt.format("%f %e", i, i);
      System.out.println(fmt);
    }
    fmt.close();

  }
}
```

It produces the following output:

```
1.230000 1.230000e+00
1.230000 1.230000e+00 123.000000 1.230000e+02
1.230000 1.230000e+00 123.000000 1.230000e+02 12300.000000
1.230000e+04
```

You can display integers in octal or hexadecimal format by using **%o** and **%x**, respectively. For example, this fragment:

```
fmt.format("Hex: %x, Octal: %o", 196, 196);
```

produces this output:

```
Hex: c4, Octal: 304
```

You can display floating-point values in hexadecimal format by using **%a**. The format produced by **%a** appears a bit strange at first glance. This is because its representation uses a form similar to scientific notation that consists of a hexadecimal significand and a decimal exponent of powers of 2. Here is the general format:

0x1.*sig*p*exp*

Here, *sig* contains the fractional portion of the significand and *exp* contains the exponent. The **p** indicates the start of the exponent. For example, this call:

```
fmt.format("%a", 512.0);
```

produces this output:

```
0x1.0p9
```

Formatting Time and Date

One of the more powerful conversion specifiers is **%t**. It lets you format time and date information. The **%t** specifier works a bit differently than the others because it requires the use of a suffix to describe the portion and precise format of the time or date desired. The suffixes are shown in Table 20-13. For example, to display minutes, you would use **%tM**, where **M** indicates minutes in a two-character field. The argument corresponding to the **%t** specifier must be of type **Calendar**, **Date**, **Long**, **long**, or **TemporalAccessor**.

Suffix	Replaced By
a	Abbreviated weekday name
A	Full weekday name
b	Abbreviated month name
B	Full month name
c	Standard date and time string formatted as *day month date hh::mm:ss tzone year*
C	First two digits of year
d	Day of month as a decimal (01—31)
D	month/day/year
e	Day of month as a decimal (1—31)
F	year-month-day
h	Abbreviated month name
H	Hour (00 to 23)
I	Hour (01 to 12)
j	Day of year as a decimal (001 to 366)
k	Hour (0 to 23)

l	Hour (1 to 12)
L	Millisecond (000 to 999)
m	Month as decimal (01 to 13)
M	Minute as decimal (00 to 59)
N	Nanosecond (000000000 to 999999999)
p	Locale's equivalent of AM or PM in lowercase
Q	Milliseconds from 1/1/1970
r	*hh:mm:ss* (12-hour format)
R	*hh:mm* (24-hour format)
S	Seconds (00 to 60)
s	Seconds from 1/1/1970 UTC
T	*hh:mm:ss* (24-hour format)
y	Year in decimal without century (00 to 99)
Y	Year in decimal including century (0001 to 9999)
z	Offset from UTC
Z	Time zone name

Table 20-13 The Time and Date Format Suffixes

Here is a program that demonstrates several of the formats:

```
// Formatting time and date.
import java.util.*;

class TimeDateFormat {
  public static void main(String args[]) {
    Formatter fmt = new Formatter();
    Calendar cal = Calendar.getInstance();

    // Display standard 12-hour time format.
    fmt.format("%tr", cal);
    System.out.println(fmt);
    fmt.close();

    // Display complete time and date information.
    fmt = new Formatter();
    fmt.format("%tc", cal);
    System.out.println(fmt);
    fmt.close();

    // Display just hour and minute.
    fmt = new Formatter();
    fmt.format("%tl:%tM", cal, cal);
    System.out.println(fmt);
    fmt.close();

    // Display month by name and number.
    fmt = new Formatter();
    fmt.format("%tB %tb %tm", cal, cal, cal);
    System.out.println(fmt);
    fmt.close();
  }
}
```

Sample output is shown here:

```
03:15:34 PM
Mon Jan 01 15:15:34 CST 2018
3:15
January Jan 01
```

The %n and %% Specifiers

The **%n** and**%%** format specifiers differ from the others in that they do not

match an argument. Instead, they are simply escape sequences that insert a character into the output sequence. The **%n** inserts a newline. The **%%** inserts a percent sign. Neither of these characters can be entered directly into the format string. Of course, you can also use the standard escape sequence **\n** to embed a newline character.

Here is an example that demonstrates the **%n** and **%%** format specifiers:

```
// Demonstrate the %n and %% format specifiers.
import java.util.*;

class FormatDemo3 {
  public static void main(String args[]) {
    Formatter fmt = new Formatter();

    fmt.format("Copying file%nTransfer is %d%% complete", 88);
    System.out.println(fmt);
    fmt.close();
  }
}
```

It displays the following output:

```
Copying file
Transfer is 88% complete
```

Specifying a Minimum Field Width

An integer placed between the **%** sign and the format conversion code acts as a *minimum field-width specifier*. This pads the output with spaces to ensure that it reaches a certain minimum length. If the string or number is longer than that minimum, it will still be printed in full. The default padding is done with spaces. If you want to pad with 0's, place a 0 before the field-width specifier. For example, **%05d** will pad a number of less than five digits with 0's so that its total length is five. The field-width specifier can be used with all format specifiers except **%n**.

The following program demonstrates the minimum field-width specifier by applying it to the **%f** conversion:

```
// Demonstrate a field-width specifier.
import java.util.*;

class FormatDemo4 {
  public static void main(String args[]) {
    Formatter fmt = new Formatter();

    fmt.format("|%f|%n|%12f|%n|%012f|",
               10.12345, 10.12345, 10.12345);
    System.out.println(fmt);
    fmt.close();

  }
}
```

This program produces the following output:

```
|10.123450|
|   10.123450|
|00010.123450|
```

The first line displays the number 10.12345 in its default width. The second line displays that value in a 12-character field. The third line displays the value in a 12-character field, padded with leading zeros.

The minimum field-width modifier is often used to produce tables in which the columns line up. For example, the next program produces a table of squares and cubes for the numbers between 1 and 10:

```
// Create a table of squares and cubes.
import java.util.*;

class FieldWidthDemo {
  public static void main(String args[]) {
    Formatter fmt;

    for(int i=1; i <= 10; i++) {
      fmt = new Formatter();
      fmt.format("%4d %4d %4d", i, i*i, i*i*i);
      System.out.println(fmt);
      fmt.close();
    }

  }
}
```

Its output is shown here:

```
 1    1    1
 2    4    8
 3    9   27
 4   16   64
 5   25  125
 6   36  216
 7   49  343
 8   64  512
 9   81  729
10  100 1000
```

Specifying Precision

A *precision specifier* can be applied to the **%f**, **%e**, **%g**, and **%s** format specifiers, among others. It follows the minimum field-width specifier (if there is one) and consists of a period followed by an integer. Its exact meaning depends upon the type of data to which it is applied.

When you apply the precision specifier to floating-point data using the **%f** or **%e** specifiers, it determines the number of decimal places displayed. For example, **%10.4f** displays a number at least ten characters wide with four decimal places. When using **%g**, the precision determines the number of significant digits. The default precision is 6.

Applied to strings, the precision specifier specifies the maximum field length.

For example, **%5.7s** displays a string of at least five and not exceeding seven characters long. If the string is longer than the maximum field width, the end characters will be truncated.

The following program illustrates the precision specifier:

```
// Demonstrate the precision modifier.
import java.util.*;

class PrecisionDemo {
  public static void main(String args[]) {
    Formatter fmt = new Formatter();

    // Format 4 decimal places.
    fmt.format("%.4f", 123.1234567);
    System.out.println(fmt);
    fmt.close();

    // Format to 2 decimal places in a 16 character field
    fmt = new Formatter();
    fmt.format("%16.2e", 123.1234567);
    System.out.println(fmt);
    fmt.close();

    // Display at most 15 characters in a string.
    fmt = new Formatter();
    fmt.format("%.15s", "Formatting with Java is now easy.");
    System.out.println(fmt);
    fmt.close();
  }
}
```

It produces the following output:

```
123.1235
        1.23e+02
Formatting with
```

Using the Format Flags

Formatter recognizes a set of format *flags* that lets you control various aspects of a conversion. All format flags are single characters, and a format flag follows the **%** in a format specification. The flags are shown here:

Flag	Effect
–	Left justification
#	Alternate conversion format
0	Output is padded with zeros rather than spaces
space	Positive numeric output is preceded by a space
+	Positive numeric output is preceded by a + sign
,	Numeric values include grouping separators
(	Negative numeric values are enclosed within parentheses

Not all flags apply to all format specifiers. The following sections explain each in detail.

Justifying Output

By default, all output is right-justified. That is, if the field width is larger than the data printed, the data will be placed on the right edge of the field. You can force output to be left-justified by placing a minus sign directly after the %. For instance, **%–10.2f** left-justifies a floating-point number with two decimal places in a 10-character field. For example, consider this program:

```
// Demonstrate left justification.
import java.util.*;

class LeftJustify {
  public static void main(String args[]) {
    Formatter fmt = new Formatter();

    // Right justify by default
    fmt.format("|%10.2f|", 123.123);
    System.out.println(fmt);
    fmt.close();

    // Now, left justify.
    fmt = new Formatter();
    fmt.format("|%-10.2f|", 123.123);
    System.out.println(fmt);
    fmt.close();
  }
}
```

It produces the following output:

```
|    123.12|
|123.12    |
```

As you can see, the second line is left-justified within a 10-character field.

The Space, +, 0, and (Flags

To cause a + sign to be shown before positive numeric values, add the + flag. For example,

```
fmt.format("%+d", 100);
```

creates this string:

```
+100
```

When creating columns of numbers, it is sometimes useful to output a space before positive values so that positive and negative values line up. To do this, add the space flag. For example,

```
// Demonstrate the space format specifiers.
import java.util.*;

class FormatDemo5 {
  public static void main(String args[]) {
    Formatter fmt = new Formatter();

    fmt.format("% d", -100);
    System.out.println(fmt);
    fmt.close();

    fmt = new Formatter();
    fmt.format("% d", 100);
    System.out.println(fmt);
    fmt.close();

    fmt = new Formatter();
    fmt.format("% d", -200);
    System.out.println(fmt);
    fmt.close();

    fmt = new Formatter();
    fmt.format("% d", 200);
    System.out.println(fmt);
    fmt.close();
  }
}
```

The output is shown here:

```
-100
 100
-200
 200
```

Notice that the positive values have a leading space, which causes the digits in the column to line up properly.

To show negative numeric output inside parentheses, rather than with a leading –, use the **(** flag. For example,

```
fmt.format("%(d", -100);
```

creates this string:

```
(100)
```

The 0 flag causes output to be padded with zeros rather than spaces.

The Comma Flag

When displaying large numbers, it is often useful to add grouping separators, which in English are commas. For example, the value 1234567 is more easily read when formatted as 1,234,567. To add grouping specifiers, use the comma **(,)** flag. For example,

```
fmt.format("%,.2f", 4356783497.34);
```

creates this string:

```
4,356,783,497.34
```

The # Flag

The **#** can be applied to **%o**, **%x**, **%e,** and **%f**. For **%e,** and **%f**, the **#** ensures that there will be a decimal point even if there are no decimal digits. If you precede the **%x** format specifier with a **#**, the hexadecimal number will be printed with a **0x** prefix. Preceding the **%o** specifier with **#** causes the number to be printed with a leading zero.

The Uppercase Option

As mentioned earlier, several of the format specifiers have uppercase versions that cause the conversion to use uppercase where appropriate. The following table describes the effect.

Specifier	Effect
%A	Causes the hexadecimal digits *a* through *f* to be displayed in uppercase as *A* through *F*. Also, the prefix **0x** is displayed as **0X**, and the **p** will be displayed as **P**.
%B	Uppercases the values **true** and **false**.
%E	Causes the *e* symbol that indicates the exponent to be displayed in uppercase.
%G	Causes the *e* symbol that indicates the exponent to be displayed in uppercase.
%H	Causes the hexadecimal digits *a* through *f* to be displayed in uppercase as *A* through *F*.
%S	Uppercases the corresponding string.
%X	Causes the hexadecimal digits *a* through *f* to be displayed in uppercase as *A* through *F*. Also, the optional prefix **0x** is displayed as **0X**, if present.

For example, this call:

```
fmt.format("%X", 250);
```

creates this string:

```
FA
```

This call:

```
fmt.format("%E", 123.1234);
```

creates this string:

```
1.231234E+02
```

Using an Argument Index

Formatter includes a very useful feature that lets you specify the argument to which a format specifier applies. Normally, format specifiers and arguments are matched in order, from left to right. That is, the first format specifier matches the first argument, the second format specifier matches the second argument, and so on. However, by using an *argument index*, you can explicitly control which argument a format specifier matches.

An argument index immediately follows the **%** in a format specifier. It has the following format:

n$

where *n* is the index of the desired argument, beginning with 1. For example, consider this example:

```
fmt.format("%3$d %1$d %2$d", 10, 20, 30);
```

It produces this string:

```
30 10 20
```

In this example, the first format specifier matches 30, the second matches 10, and the third matches 20. Thus, the arguments are used in an order other than strictly left to right.

One advantage of argument indexes is that they enable you to reuse an argument without having to specify it twice. For example, consider this line:

```
fmt.format("%d in hex is %1$x", 255);
```

It produces the following string:

```
255 in hex is ff
```

As you can see, the argument 255 is used by both format specifiers.

There is a convenient shorthand called a *relative index* that enables you to reuse the argument matched by the preceding format specifier. Simply specify < for the argument index. For example, the following call to **format( )** produces the same results as the previous example:

```
fmt.format("%d in hex is %<x", 255);
```

Relative indexes are especially useful when creating custom time and date formats. Consider the following example:

```
// Use relative indexes to simplify the
// creation of a custom time and date format.
import java.util.*;

class FormatDemo6 {
  public static void main(String args[]) {
    Formatter fmt = new Formatter();
    Calendar cal = Calendar.getInstance();

    fmt.format("Today is day %te of %<tB, %<tY", cal);
    System.out.println(fmt);
    fmt.close();
  }
}
```

Here is sample output:

```
Today is day 1 of January, 2018
```

Because of relative indexing, the argument **cal** need only be passed once, rather than three times.

Closing a Formatter

In general, you should close a **Formatter** when you are done using it. Doing so frees any resources that it was using. This is especially important when formatting to a file, but it can be important in other cases, too. As the previous examples have shown, one way to close a **Formatter** is to explicitly call **close( )**. However, **Formatter** also implements the **AutoCloseable** interface. This means that it supports the **try**-with-resources statement. Using this approach, the **Formatter** is automatically closed when it is no longer needed.

The **try**-with-resources statement is described in Chapter 13, in connection with files, because files are some of the most commonly used resources that must be closed. However, the same basic techniques apply here. For example, here is the first **Formatter** example reworked to use automatic resource management:

```
// Use automatic resource management with Formatter.
import java.util.*;

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

    try (Formatter fmt = new Formatter())
    {
      fmt.format("Formatting %s is easy %d %f", "with Java",
                 10, 98.6);
      System.out.println(fmt);
    }
  }
}
```

The output is the same as before.

The Java printf( ) Connection

Although there is nothing technically wrong with using **Formatter** directly (as the preceding examples have done) when creating output that will be displayed on the console, there is a more convenient alternative: the **printf( )** method. The **printf( )** method automatically uses **Formatter** to create a formatted string. It then displays that string on **System.out**, which is the console by default. The **printf( )** method is defined by both **PrintStream** and **PrintWriter**. The **printf( )** method is described in Chapter 21.

Scanner

Scanner is the complement of **Formatter**. It reads formatted input and converts it into its binary form. **Scanner** can be used to read input from the console, a file, a string, or any source that implements the **Readable** interface or **ReadableByteChannel**. For example, you can use **Scanner** to read a number from the keyboard and assign its value to a variable. As you will see, given its power, **Scanner** is surprisingly easy to use.

The Scanner Constructors

Scanner defines many constructors. A sampling is shown in Table 20-14. In general, a **Scanner** can be created for a **String**, an **InputStream**, a **File**, a **Path**,

or any object that implements the **Readable** or **ReadableByteChannel** interfaces. Here are some examples.

Method	Description
Scanner(File *from*) throws FileNotFoundException	Creates a **Scanner** that uses the file specified by *from* as a source for input.
Scanner(File *from*, String *charset*) throws FileNotFoundException	Creates a **Scanner** that uses the file specified by *from* with the encoding specified by *charset* as a source for input.
Scanner(InputStream *from*)	Creates a **Scanner** that uses the stream specified by *from* as a source for input.
Scanner(InputStream *from*, String *charset*)	Creates a **Scanner** that uses the stream specified by *from* with the encoding specified by *charset* as a source for input.
Scanner(Path *from*) throws IOException	Creates a **Scanner** that uses the file specified by *from* as a source for input.
Scanner(Path *from*, String *charset*) throws IOException	Creates a **Scanner** that uses the file specified by *from* with the encoding specified by *charset* as a source for input.
Scanner(Readable *from*)	Creates a **Scanner** that uses the **Readable** object specified by *from* as a source for input.
Scanner (ReadableByteChannel *from*)	Creates a **Scanner** that uses the **ReadableByteChannel** specified by *from* as a source for input.
Scanner(ReadableByteChannel *from*, String *charset*)	Creates a **Scanner** that uses the **ReadableByteChannel** specified by *from* with the encoding specified by *charset* as a source for input.
Scanner(String *from*)	Creates a **Scanner** that uses the string specified by *from* as a source for input.

Table 20-14 A Sampling of **Scanner** Constructors

The following sequence creates a **Scanner** that reads the file **Test.txt**:

```
FileReader fin = new FileReader("Test.txt");
Scanner src = new Scanner(fin);
```

This works because **FileReader** implements the **Readable** interface. Thus, the call to the constructor resolves to **Scanner(Readable)**.

This next line creates a **Scanner** that reads from standard input, which is the keyboard by default:

```
Scanner conin = new Scanner(System.in);
```

This works because **System.in** is an object of type **InputStream**. Thus, the call to the constructor maps to **Scanner(InputStream)**.

The next sequence creates a **Scanner** that reads from a string.

```
String instr = "10 99.88 scanning is easy.";
Scanner conin = new Scanner(instr);
```

Scanning Basics

Once you have created a **Scanner**, it is a simple matter to use it to read formatted input. In general, a **Scanner** reads *tokens* from the underlying source that you specified when the **Scanner** was created. As it relates to **Scanner**, a token is a portion of input that is delineated by a set of delimiters, which is whitespace by default. A token is read by matching it with a particular *regular expression*, which defines the format of the data. Although **Scanner** allows you to define the specific type of expression that its next input operation will match, it includes many predefined patterns, which match the primitive types, such as **int** and **double**, and strings. Thus, often you won't need to specify a pattern to match.

In general, to use **Scanner**, follow this procedure:

1. Determine if a specific type of input is available by calling one of **Scanner**'s **hasNext***X* methods, where *X* is the type of data desired.
2. If input is available, read it by calling one of **Scanner**'s **next***X* methods.
3. Repeat the process until input is exhausted.
4. Close the **Scanner** by calling **close( )**.

As the preceding indicates, **Scanner** defines two sets of methods that enable you to read input. The first are the **hasNext***X* methods, which are shown in Table 20-15. These methods determine if the specified type of input is available. For example, calling **hasNextInt( )** returns **true** only if the next token to be read is an integer. If the desired data is available, then you read it by calling one of **Scanner**'s **next***X* methods, which are shown in Table 20-16. For example, to read the next integer, call **nextInt( )**. The following sequence shows how to read a list of integers from the keyboard.

Method	Description
boolean hasNext( )	Returns **true** if another token of any type is available to be read. Returns **false** otherwise.
boolean hasNext(Pattern *pattern*)	Returns **true** if a token that matches the pattern passed in *pattern* is available to be read. Returns **false** otherwise.
boolean hasNext(String *pattern*)	Returns **true** if a token that matches the pattern passed in *pattern* is available to be read. Returns **false** otherwise.
boolean hasNextBigDecimal( )	Returns **true** if a value that can be stored in a **BigDecimal** object is available to be read. Returns **false** otherwise.
boolean hasNextBigInteger( )	Returns **true** if a value that can be stored in a **BigInteger** object is available to be read. Returns **false** otherwise. The default radix is used. (Unless changed, the default radix is 10.)
boolean hasNextBigInteger(int *radix*)	Returns **true** if a value in the specified radix that can be stored in a **BigInteger** object is available to be read. Returns **false** otherwise.
boolean hasNextBoolean( )	Returns **true** if a **boolean** value is available to be read. Returns **false** otherwise.
boolean hasNextByte( )	Returns **true** if a **byte** value is available to be read. Returns **false** otherwise. The default radix is used. (Unless changed, the default radix is 10.)
boolean hasNextByte(int *radix*)	Returns **true** if a **byte** value in the specified radix is available to be read. Returns **false** otherwise.
boolean hasNextDouble( )	Returns **true** if a **double** value is available to be read. Returns **false** otherwise.
boolean hasNextFloat( )	Returns **true** if a **float** value is available to be read. Returns **false** otherwise.
boolean hasNextInt( )	Returns **true** if an **int** value is available to be read. Returns **false** otherwise. The default radix is used. (Unless changed, the default radix is 10.)
boolean hasNextInt(int *radix*)	Returns **true** if an **int** value in the specified radix is available to be read. Returns **false** otherwise.
boolean hasNextLine( )	Returns **true** if a line of input is available.
boolean hasNextLong( )	Returns **true** if a **long** value is available to be read. Returns **false** otherwise. The default radix is used. (Unless changed, the default radix is 10.)
boolean hasNextLong(int *radix*)	Returns **true** if a **long** value in the specified radix is available to be read. Returns **false** otherwise.
boolean hasNextShort( )	Returns **true** if a **short** value is available to be read. Returns **false** otherwise. The default radix is used. (Unless changed, the default radix is 10.)
boolean hasNextShort(int *radix*)	Returns **true** if a **short** value in the specified radix is available to be read. Returns **false** otherwise.

Table 20-15 The **Scanner hasNext** Methods

Method	Description
String next( )	Returns the next token of any type from the input source.
String next(Pattern *pattern*)	Returns the next token that matches the pattern passed in *pattern* from the input source.
String next(String *pattern*)	Returns the next token that matches the pattern passed in *pattern* from the input source.
BigDecimal nextBigDecimal( )	Returns the next token as a **BigDecimal** object.
BigInteger nextBigInteger( )	Returns the next token as a **BigInteger** object. The default radix is used. (Unless changed, the default radix is 10.)
BigInteger nextBigInteger(int *radix*)	Returns the next token (using the specified radix) as a **BigInteger** object.
boolean nextBoolean( )	Returns the next token as a **boolean** value.
byte nextByte( )	Returns the next token as a **byte** value. The default radix is used. (Unless changed, the default radix is 10.)
byte nextByte(int *radix*)	Returns the next token (using the specified radix) as a **byte** value.
double nextDouble( )	Returns the next token as a **double** value.
float nextFloat( )	Returns the next token as a **float** value.
int nextInt( )	Returns the next token as an **int** value. The default radix is used. (Unless changed, the default radix is 10.)
int nextInt(int *radix*)	Returns the next token (using the specified radix) as an **int** value.
String nextLine( )	Returns the next line of input as a string.
long nextLong( )	Returns the next token as a **long** value. The default radix is used. (Unless changed, the default radix is 10.)
long nextLong(int *radix*)	Returns the next token (using the specified radix) as a **long** value.
short nextShort( )	Returns the next token as a **short** value. The default radix is used. (Unless changed, the default radix is 10.)
short nextShort(int *radix*)	Returns the next token (using the specified radix) as a **short** value.

Table 20-16 The **Scanner next** Methods

```
Scanner conin = new Scanner(System.in);
int i;

// Read a list of integers.
while(conin.hasNextInt()) {
  i = conin.nextInt();
  // ...
}
```

The **while** loop stops as soon as the next token is not an integer. Thus, the loop stops reading integers as soon as a non-integer is encountered in the input stream.

If a **next** method cannot find the type of data it is looking for, it throws an **InputMismatchException**. A **NoSuchElementException** is thrown if no more input is available. For this reason, it is best to first confirm that the desired type of data is available by calling a **hasNext** method before calling its corresponding **next** method.

Some Scanner Examples

Scanner makes what could be a tedious task into an easy one. To understand why, let's look at some examples. The following program averages a list of numbers entered at the keyboard:

```
// Use Scanner to compute an average of the values.
import java.util.*;

class AvgNums {
  public static void main(String args[]) {
    Scanner conin = new Scanner(System.in);

    int count = 0;
    double sum = 0.0;

    System.out.println("Enter numbers to average.");

    // Read and sum numbers.
    while(conin.hasNext()) {
      if(conin.hasNextDouble()) {
        sum += conin.nextDouble();
        count++;
      }
      else {
        String str = conin.next();
        if(str.equals("done")) break;
        else {
          System.out.println("Data format error.");
          return;
        }
      }
    }

    conin.close();
    System.out.println("Average is " + sum / count);
  }
}
```

The program reads numbers from the keyboard, summing them in the process, until the user enters the string "done". It then stops input and displays the average of the numbers. Here is a sample run:

```
Enter numbers to average.
1.2
2
3.4
4
done
Average is 2.65
```

The program reads numbers until it encounters a token that does not represent a valid **double** value. When this occurs, it confirms that the token is the string "done". If it is, the program terminates normally. Otherwise, it displays an error.

Notice that the numbers are read by calling **nextDouble( )**. This method reads any number that can be converted into a **double** value, including an integer value, such as 2, and a floating-point value like 3.4. Thus, a number read by **nextDouble( )** need not specify a decimal point. This same general principle applies to all **next** methods. They will match and read any data format that can represent the type of value being requested.

One thing that is especially nice about **Scanner** is that the same technique used to read from one source can be used to read from another. For example, here is the preceding program reworked to average a list of numbers contained in a text file:

```
// Use Scanner to compute an average of the values in a file.
import java.util.*;
import java.io.*;

class AvgFile {
  public static void main(String args[])
    throws IOException {

    int count = 0;
    double sum = 0.0;

    // Write output to a file.
    FileWriter fout = new FileWriter("test.txt");
    fout.write("2 3.4 5 6 7.4 9.1 10.5 done");
    fout.close();

    FileReader fin = new FileReader("Test.txt");

    Scanner src = new Scanner(fin);

    // Read and sum numbers.
    while(src.hasNext()) {
      if(src.hasNextDouble()) {
        sum += src.nextDouble();
        count++;
      }

      else {
        String str = src.next();
        if(str.equals("done")) break;
        else {
          System.out.println("File format error.");
          return;
        }
      }
    }

    src.close();
    System.out.println("Average is " + sum / count);
  }
}
```

Here is the output:

```
Average is 6.2
```

The preceding program illustrates another important feature of **Scanner**. Notice that the file reader referred to by **fin** is not closed directly. Rather, it is closed automatically when **src** calls **close( )**. When you close a **Scanner**, the **Readable** associated with it is also closed (if that **Readable** implements the **Closeable** interface). Therefore, in this case, the file referred to by **fin** is automatically closed when **src** is closed.

Scanner also implements the **AutoCloseable** interface. This means that it can be managed by a **try**-with-resources block. As explained in Chapter 13, when **try**-with-resources is used, the scanner is automatically closed when the block ends. For example, **src** in the preceding program could have been managed like this:

```
try (Scanner src = new Scanner(fin))
{
  // Read and sum numbers.
  while(src.hasNext()) {
    if(src.hasNextDouble()) {
      sum += src.nextDouble();
      count++;
    }
    else {
      String str = src.next();
      if(str.equals("done")) break;
      else {
        System.out.println("File format error.");
        return;
      }
    }
  }
}
```

To clearly demonstrate the closing of a **Scanner**, the following examples will call **close( )** explicitly, but you should feel free to use **try**-with-resources in your own code when appropriate.

One other point: To keep this and the other examples in this section compact, I/O exceptions are simply thrown out of **main( )**. However, your real-world code will normally handle I/O exceptions itself.

You can use **Scanner** to read input that contains several different types of data—even if the order of that data is unknown in advance. You must simply

check what type of data is available before reading it. For example, consider this program:

```
// Use Scanner to read various types of data from a file.
import java.util.*;
import java.io.*;

class ScanMixed {
  public static void main(String args[])
    throws IOException {

    int i;
    double d;
    boolean b;
    String str;

    // Write output to a file.
    FileWriter fout = new FileWriter("test.txt");
    fout.write("Testing Scanner 10 12.2 one true two false");
    fout.close();

    FileReader fin = new FileReader("Test.txt");

    Scanner src = new Scanner(fin);

    // Read to end.
    while(src.hasNext()) {
      if(src.hasNextInt()) {
        i = src.nextInt();
        System.out.println("int: " + i);
      }
      else if(src.hasNextDouble()) {
        d = src.nextDouble();
        System.out.println("double: " + d);
      }
      else if(src.hasNextBoolean()) {
        b = src.nextBoolean();
        System.out.println("boolean: " + b);
      }
      else {
        str = src.next();
        System.out.println("String: " + str);
      }
    }

    src.close();
  }
}
```

Here is the output:

```
String: Testing
String: Scanner
int: 10
double: 12.2
String: one
boolean: true
String: two
boolean: false
```

When reading mixed data types, as the preceding program does, you need to be a bit careful about the order in which you call the **next** methods. For example, if the loop reversed the order of the calls to **nextInt( )** and **nextDouble( )**, both numeric values would have been read as **double**s, because **nextDouble( )** matches any numeric string that can be represented as a **double**.

Setting Delimiters

Scanner defines where a token starts and ends based on a set of *delimiters*. The default delimiters are the whitespace characters, and this is the delimiter set that the preceding examples have used. However, it is possible to change the delimiters by calling the **useDelimiter( )** method, shown here:

Scanner useDelimiter(String *pattern*)

Scanner useDelimiter(Pattern *pattern*)

Here, *pattern* is a regular expression that specifies the delimiter set.

Here is the program that reworks the average program shown earlier so that it reads a list of numbers that are separated by commas, and any number of spaces:

```
// Use Scanner to compute an average a list of
// comma-separated values.
import java.util.*;
import java.io.*;

class SetDelimiters {
  public static void main(String args[])
    throws IOException {

    int count = 0;
    double sum = 0.0;

    // Write output to a file.
    FileWriter fout = new FileWriter("test.txt");

    // Now, store values in comma-separated list.
    fout.write("2, 3.4,     5,6, 7.4, 9.1, 10.5, done");
    fout.close();

    FileReader fin = new FileReader("Test.txt");

    Scanner src = new Scanner(fin);

    // Set delimiters to space and comma.
    src.useDelimiter(", *");

    // Read and sum numbers.
    while(src.hasNext()) {
      if(src.hasNextDouble()) {
        sum += src.nextDouble();
        count++;
      }
      else {
        String str = src.next();
        if(str.equals("done")) break;
        else {
          System.out.println("File format error.");
          return;
        }
      }
    }

    src.close();
    System.out.println("Average is " + sum / count);
  }
}
```

In this version, the numbers written to **test.txt** are separated by commas and spaces. The use of the delimiter pattern **", * "** tells **Scanner** to match a comma and zero or more spaces as delimiters. The output is the same as before.

You can obtain the current delimiter pattern by calling **delimiter( )**, shown here:

Pattern delimiter( )

Other Scanner Features

Scanner defines several other methods in addition to those already discussed. One that is particularly useful in some circumstances is **findInLine( )**. Its general forms are shown here:

String findInLine(Pattern *pattern*)
String findInLine(String *pattern*)

This method searches for the specified pattern within the next line of text. If the pattern is found, the matching token is consumed and returned. Otherwise, **null** is returned. It operates independently of any delimiter set. This method is useful if you want to locate a specific pattern. For example, the following program locates the Age field in the input string and then displays the age:

```
// Demonstrate findInLine().
import java.util.*;

class FindInLineDemo {
  public static void main(String args[]) {
    String instr = "Name: Tom Age: 28 ID: 77";

    Scanner conin = new Scanner(instr);

    // Find and display age.
    conin.findInLine("Age:"); // find Age

    if(conin.hasNext())
      System.out.println(conin.next());
    else
      System.out.println("Error!");

    conin.close();
  }
}
```

The output is **28**. In the program, **findInLine( )** is used to find an occurrence of the pattern "Age". Once found, the next token is read, which is the age.

Related to **findInLine( )** is **findWithinHorizon( )**. It is shown here:

String findWithinHorizon(Pattern *pattern*, int *count*)

String findWithinHorizon(String *pattern*, int *count*)

This method attempts to find an occurrence of the specified pattern within the next *count* characters. If successful, it returns the matching pattern. Otherwise, it returns **null**. If *count* is zero, then all input is searched until either a match is found or the end of input is encountered.

You can bypass a pattern using **skip( )**, shown here:

Scanner skip(Pattern *pattern*)

Scanner skip(String *pattern*)

If *pattern* is matched, **skip( )** simply advances beyond it and returns a reference to the invoking object. If pattern is not found, **skip( )** throws **NoSuchElementException**.

Other **Scanner** methods include **radix( )**, which returns the default radix used

by the **Scanner**; **useRadix( )**, which sets the radix; **reset( )**, which resets the scanner; and **close( )**, which closes the scanner. JDK 9 added the methods **tokens( )**, which returns all tokens in the form of a **Stream<String>**, and **findAll( )**, which returns tokens that match the specified pattern in the form of a **Stream<MatchResult>**.

The ResourceBundle, ListResourceBundle, and PropertyResourceBundle Classes

The **java.util** package includes three classes that aid in the internationalization of your program. The first is the abstract class **ResourceBundle**. It defines methods that enable you to manage a collection of locale-sensitive resources, such as the strings that are used to label the user interface elements in your program. You can define two or more sets of translated strings that support various languages, such as English, German, or Chinese, with each translation set residing in its own bundle. You can then load the bundle appropriate to the current locale and use the strings to construct the program's user interface.

Resource bundles are identified by their *family name* (also called their *base name*). To the family name can be added a two-character lowercase *language code* which specifies the language. In this case, if a requested locale matches the language code, then that version of the resource bundle is used. For example, a resource bundle with a family name of **SampleRB** could have a German version called **SampleRB_de** and a Russian version called **SampleRB_ru**. (Notice that an underscore links the family name to the language code.) Therefore, if the locale is **Locale.GERMAN**, **SampleRB_de** will be used.

It is also possible to indicate specific variants of a language that relate to a specific country by specifying a *country code* after the language code. A country code is a two-character uppercase identifier, such as **AU** for Australia or **IN** for India. A country code is also preceded by an underscore when linked to the resource bundle name. Other variations are also supported. A resource bundle that has only the family name is the default bundle. It is used when no language-specific bundles are applicable.

NOTE The language codes are defined by ISO standard 639 and the country codes by ISO standard 3166.

The methods defined by **ResourceBundle** are summarized in Table 20-17. One important point: **null** keys are not allowed and several of the methods will throw a **NullPointerException** if **null** is passed as the key. Notice the nested

class **ResourceBundle.Control**. It is used to control the resource-bundle loading process.

Method	Description
static final void clearCache( )	Deletes all resource bundles from the cache that were loaded by the class loader. Beginning with JDK 9, this method deletes all resource bundles from the cache that were loaded by the module from which this method is called.
static final void clearCache(ClassLoader *ldr*)	Deletes all resource bundles from the cache that were loaded by *ldr*.
boolean containsKey(String *k*)	Returns **true** if *k* is a key within the invoking resource bundle (or its parent).
String getBaseBundleName( )	Returns the resource bundle's base name if available. Returns **null** otherwise.
static final ResourceBundle getBundle(String *familyName*)	Loads the resource bundle with a family name of *familyName* using the default locale. Throws **MissingResourceException** if no resource bundle matching the name is available.
static ResourceBundle getBundle(String *familyName*, Module *mod*)	Loads the resource bundle with a family name of *familyName* for the module specified by *mod*. The default locale is used. Throws **MissingResourceException** if no resource bundle matching the name is available.
static final ResourceBundle getBundle(String *familyName*, Locale *loc*)	Loads the resource bundle with a family name of *familyName* using the specified locale. Throws **MissingResourceException** if no resource bundle matching the name is available.
static ResourceBundle getBundle(String *familyName*, Locale *loc*, Module *mod*)	Loads the resource bundle with a family name of *familyName* using the locale passed in *loc* for the module specified by *mod*. Throws **MissingResourceException** if no resource bundle matching the name is available.

static ResourceBundle getBundle(String *familyName*, Locale *loc*, ClassLoader *ldr*)	Loads the resource bundle with a family name of *familyName* using the specified locale and the specified class loader. Throws **MissingResourceException** if no resource bundle matching the name is available.
static final ResourceBundle getBundle(String *familyName*, ResourceBundle.Control *cntl*)	Loads the resource bundle with a family name of *familyName* using the default locale. The loading process is under the control of *cntl*. Throws **MissingResourceException** if no resource bundle matching the name is available.
static final ResourceBundle getBundle(String *familyName*, Locale *loc*, ResourceBundle.Control *cntl*)	Loads the resource bundle with a family name of *familyName* using the specified locale. The loading process is under the control of *cntl*. Throws **MissingResourceException** if no resource bundle matching the name is available.
static ResourceBundle getBundle(String *familyName*, Locale *loc*, ClassLoader *ldr*, ResourceBundle.Control *cntl*)	Loads the resource bundle with a family name of *familyName* using the specified locale and the specified class loader. The loading process is under the control of *cntl*. Throws **MissingResourceException** if no resource bundle matching the name is available.
abstract Enumeration<String> getKeys( )	Returns the resource bundle keys as an enumeration of strings. Any parent's keys are also obtained.
Locale getLocale( )	Returns the locale supported by the resource bundle.
final Object getObject(String *k*)	Returns the object associated with the key passed via *k*. Throws **MissingResourceException** if *k* is not in the resource bundle.

final String getString(String *k*)	Returns the string associated with the key passed via *k*. Throws **MissingResourceException** if *k* is not in the resource bundle. Throws **ClassCastException** if the object associated with *k* is not a string.
final String[ ] getStringArray(String *k*)	Returns the string array associated with the key passed via *k*. Throws **MissingResourceException** if *k* is not in the resource bundle. Throws **MissingResourceException** if the object associated with *k* is not a string array.
protected abstract Object handleGetObject(String *k*)	Returns the object associated with the key passed via *k*. Returns **null** if *k* is not in the resource bundle.
protected Set<String> handleKeySet( )	Returns the resource bundle keys as a set of strings. No parent's keys are obtained.
Set<String> keySet( )	Returns the resource bundle keys as a set of strings. Any parent keys are also obtained.
protected void setParent(ResourceBundle *parent*)	Sets *parent* as the parent bundle for the resource bundle. When a key is looked up, the parent will be searched if the key is not found in the invoking resource object.

Table 20-17 The Methods Defined by **ResourceBundle**

NOTE Notice that JDK 9 added methods to **ResourceBundle** that support modules. Furthermore, the addition of modules raises several issues related to the use of resource bundles that are beyond the scope of this discussion. Consult the API documentation for details on how modules affect the use of **ResourceBundle**.

There are two subclasses of **ResourceBundle**. The first is **PropertyResourceBundle**, which manages resources by using property files. **PropertyResourceBundle** adds no methods of its own. The second is the abstract class **ListResourceBundle**, which manages resources in an array of key/value pairs. **ListResourceBundle** adds the method **getContents( )**, which all subclasses must implement. It is shown here:

```
protected abstract Object[ ][ ] getContents( )
```

It returns a two-dimensional array that contains key/value pairs that represent resources. The keys must be strings. The values are typically strings, but can be other types of objects.

Here is an example that demonstrates using a resource bundle in an unnamed module. The resource bundle has the family name **SampleRB**. Two resource bundle classes of this family are created by extending **ListResourceBundle**. The

first is called **SampleRB**, and it is the default bundle (which uses English). It is shown here:

```
import java.util.*;
public class SampleRB extends ListResourceBundle {
  protected Object[][] getContents() {
    Object[][] resources = new Object[3][2];

    resources[0][0] = "title";
    resources[0][1] = "My Program";

    resources[1][0] = "StopText";
    resources[1][1] = "Stop";

    resources[2][0] = "StartText";
    resources[2][1] = "Start";

    return resources;
  }
}
```

The second resource bundle, shown next, is called **SampleRB_de**. It contains the German translation.

```
import java.util.*;

// German version.
public class SampleRB_de extends ListResourceBundle {
  protected Object[][] getContents() {
    Object[][] resources = new Object[3][2];

    resources[0][0] = "title";
    resources[0][1] = "Mein Programm";

    resources[1][0] = "StopText";
    resources[1][1] = "Anschlag";

    resources[2][0] = "StartText";
    resources[2][1] = "Anfang";

    return resources;
  }
}
```

The following program demonstrates these two resource bundles by displaying the string associated with each key for both the default (English) version and the German version:

```
// Demonstrate a resource bundle.
import java.util.*;

class LRBDemo {
  public static void main(String args[]) {
    // Load the default bundle.
    ResourceBundle rd = ResourceBundle.getBundle("SampleRB");

    System.out.println("English version: ");
    System.out.println("String for Title key : " +
                       rd.getString("title"));

    System.out.println("String for StopText key: " +
                       rd.getString("StopText"));

    System.out.println("String for StartText key: " +
                       rd.getString("StartText"));

    // Load the German bundle.
    rd = ResourceBundle.getBundle("SampleRB", Locale.GERMAN);

    System.out.println("\nGerman version: ");
    System.out.println("String for Title key : " +
                       rd.getString("title"));

    System.out.println("String for StopText key: " +
                       rd.getString("StopText"));

    System.out.println("String for StartText key: " +
                       rd.getString("StartText"));
  }
}
```

The output from the program is shown here:

```
English version:
String for Title key : My Program
String for StopText key: Stop
String for StartText key: Start
```

```
German version:
String for Title key : Mein Programm
String for StopText key: Anschlag
String for StartText key: Anfang
```

Miscellaneous Utility Classes and Interfaces

In addition to the classes already discussed, **java.util** includes the following classes:

Base64	Supports Base64 encoding. **Encoder** and **Decoder** nested classes are also defined.
DoubleSummaryStatistics	Supports the compilation of **double** values. The following statistics are available: average, minimum, maximum, count, and sum.
EventListenerProxy	Extends the **EventListener** class to allow additional parameters. See Chapter 24 for a discussion of event listeners.
EventObject	The superclass for all event classes. Events are discussed in Chapter 24.
FormattableFlags	Defines formatting flags that are used with the **Formattable** interface.
IntSummaryStatistics	Supports the compilation of **int** values. The following statistics are available: average, minimum, maximum, count, and sum.
Objects	Various methods that operate on objects.
PropertyPermission	Manages property permissions.
ServiceLoader	Provides a means of finding service providers.
StringJoiner	Supports the concatenation of **CharSequence**s, which may include a separator, a prefix, and a suffix.
UUID	Encapsulates and manages Universally Unique Identifiers (UUIDs).

The following interfaces are also packaged in **java.util**:

EventListener	Indicates that a class is an event listener. Events are discussed in Chapter 24.
Formattable	Enables a class to provide custom formatting.

The java.util Subpackages

Java defines the following subpackages of **java.util**:

- java.util.concurrent
- java.util.concurrent.atomic
- java.util.concurrent.locks
- java.util.function
- java.util.jar
- java.util.logging
- java.util.prefs
- java.util.regex
- java.util.spi
- java.util.stream
- java.util.zip

Except as otherwise noted, all are part of the **java.base** module. Each is briefly examined here.

java.util.concurrent, java.util.concurrent.atomic, and java.util.concurrent.locks

The **java.util.concurrent** package along with its two subpackages, **java.util.concurrent.atomic** and **java.util.concurrent.locks**, support concurrent programming. These packages provide a high-performance alternative to using Java's built-in synchronization features when thread-safe operation is required. The **java.util.concurrent** package also provides the Fork/Join Framework. These packages are examined in detail in Chapter 28.

java.util.function

The **java.util.function** package defines several predefined functional interfaces that you can use when creating lambda expressions or method references. They are also widely used throughout the Java API. The functional interfaces defined by **java.util.function** are shown in Table 20-18 along with a synopsis of their abstract methods. Be aware that some of these interfaces also define default or static methods that supply additional functionality. You will want to explore them fully on your own. (For a discussion of the use of functional interfaces, see

Chapter 15.)

Interface	Abstract Method
BiConsumer<T, U>	void accept(T *tVal*, U *uVal*) **Description**: Acts on *tVal* and *uVal*.
BiFunction<T, U, R>	R apply(T *tVal*, U *uVal*) **Description**: Acts on *tVal* and *uVal* and returns the result.
BinaryOperator<T>	T apply(T *val1*, T *val2*) **Description**: Acts on two objects of the same type and returns the result, which is also of the same type.
BiPredicate<T, U>	boolean test(T *tVal*, U *uVal*) **Description**: Returns **true** if *tVal* and *uVal* satisfy the condition defined by **test( )** and **false** otherwise.
BooleanSupplier	boolean getAsBoolean( ) **Description**: Returns a **boolean** value.
Consumer<T>	void accept(T *val*) **Description**: Acts on *val*.
DoubleBinaryOperator	double applyAsDouble(double *val1*, double *val2*) **Description**: Acts on two **double** values and returns a **double** result.
DoubleConsumer	void accept(double *val*) **Description**: Acts on *val*.
DoubleFunction<R>	R apply(double *val*) **Description**: Acts on a **double** value and returns the result.
DoublePredicate	boolean test(double *val*) **Description**: Returns **true** if *val* satisfies the condition defined by **test( )** and **false** otherwise.
DoubleSupplier	double getAsDouble( ) **Description**: Returns a **double** result.

DoubleToIntFunction	int applyAsInt(double *val*) **Description**: Acts on a **double** value and returns the result as an **int**.
DoubleToLongFunction	long applyAsLong(double *val*) **Description**: Acts on a **double** value and returns the result as a **long**.
DoubleUnaryOperator	double applyAsDouble(double *val*) **Description**: Acts on a **double** and returns a **double** result.
Function<T, R>	R apply(T *val*) **Description**: Acts on *val* and returns the result.
IntBinaryOperator	int applyAsInt(int *val1*, int *val2*) **Description**: Acts on two **int** values and returns an **int** result.
IntConsumer	int accept(int *val*) **Description**: Acts on *val*.
IntFunction<R>	R apply(int *val*) **Description**: Acts on an **int** value and returns the result.
IntPredicate	boolean test(int *val*) **Description**: Returns **true** if *val* satisfies the condition defined by **test( )** and **false** otherwise.
IntSupplier	int getAsInt( ) **Description**: Returns an **int** result.
IntToDoubleFunction	double applyAsDouble(int *val*) **Description**: Acts on an **int** value and returns the result as a **double**.
IntToLongFunction	long applyAsLong(int *val*) **Description**: Acts on an **int** value and returns the result as a **long**.
IntUnaryOperator	int applyAsInt(int *val*) **Description**: Acts on an **int** and returns an **int** result.

LongBinaryOperator	long applyAsLong(long *val1*, long *val2*) **Description**: Acts on two **long** values and returns a **long** result.
LongConsumer	void accept(long *val*) **Description**: Acts on *val*.
LongFunction<R>	R apply(long *val*) **Description**: Acts on a **long** value and returns the result.
LongPredicate	boolean test(long *val*) **Description**: Returns **true** if *val* satisfies the condition defined by **test( )** and **false** otherwise.
LongSupplier	long getAsLong( ) **Description**: Returns a **long** result.
LongToDoubleFunction	double applyAsDouble(long *val*) **Description**: Acts on a **long** value and returns the result as a **double**.
LongToIntFunction	int applyAsInt(long *val*) **Description**: Acts on a **long** value and returns the result as an **int**.
LongUnaryOperator	long applyAsLong(long *val*) **Description**: Acts on a **long** and returns a **long** result.
ObjDoubleConsumer<T>	void accept(T *val1*, double *val2*) **Description**: Acts on *val1* and the **double** value *val2*.
ObjIntConsumer<T>	void accept(T *val1*, int *val2*) **Description**: Acts on *val1* and the **int** value *val2*.
ObjLongConsumer<T>	void accept(T *val1*, long *val2*) **Description**: Acts on *val1* and the **long** value *val2*.
Predicate<T>	boolean test(T *val*) **Description**: Returns **true** if *val* satisfies the condition defined by **test( )** and **false** otherwise.

Supplier<T>	T get( ) **Description**: Returns an object of type **T**.
ToDoubleBiFunction<T, U>	double applyAsDouble(T *tVal*, U *uVal*) **Description**: Acts on *tVal* and *uVal* and returns the result as a **double**.
ToDoubleFunction<T>	double applyAsDouble(T *val*) **Description**: Acts on *val* and returns the result as a **double**.
ToIntBiFunction<T, U>	int applyAsInt(T *tVal*, U *uVal*) **Description**: Acts on *tVal* and *uVal* and returns the result as an **int**.
ToIntFunction<T>	int applyAsInt(T *val*) **Description**: Acts on *val* and returns the result as an **int**.
ToLongBiFunction<T, U>	long applyAsLong(T *tVal*, U *uVal*) **Description**: Acts on *tVal* and *uVal* and returns the result as a **long**.
ToLongFunction<T>	long applyAsLong(T *val*) **Description**: Acts on *val* and returns the result as a **long**.
UnaryOperator<T>	T apply(T *val*) **Description**: Acts on *val* and returns the result

Table 20-18 Functional Interfaces Defined by **java.util.function** and Their Abstract Methods

java.util.jar

The **java.util.jar** package provides the ability to read and write Java Archive (JAR) files.

java.util.logging

The **java.util.logging** package provides support for program activity logs, which can be used to record program actions, and to help find and debug problems. This package is in the **java.logging** module.

java.util.prefs

The **java.util.prefs** package provides support for user preferences. It is typically used to support program configuration. This package is in the **java.prefs**

module.

java.util.regex

The **java.util.regex** package provides support for regular expression handling. It is described in detail in Chapter 30.

java.util.spi

The **java.util.spi** package provides support for service providers.

java.util.stream

The **java.util.stream** package contains Java's stream API. A discussion of the stream API is found in Chapter 29.

java.util.zip

The **java.util.zip** package provides the ability to read and write files in the popular ZIP and GZIP formats. Both ZIP and GZIP input and output streams are available.

CHAPTER

21

Input/Output: Exploring java.io

This chapter explores **java.io**, which provides support for I/O operations. Chapter 13 presented an overview of Java's I/O system, including basic techniques for reading and writing files, handling I/O exceptions, and closing a file. Here, we will examine the Java I/O system in greater detail.

As all programmers learn early on, most programs cannot accomplish their goals without accessing external data. Data is retrieved from an *input* source. The results of a program are sent to an *output* destination. In Java, these sources or destinations are defined very broadly. For example, a network connection, memory buffer, or disk file can be manipulated by the Java I/O classes. Although physically different, these devices are all handled by the same abstraction: the *stream*. An I/O stream, as explained in Chapter 13, is a logical entity that either produces or consumes information. An I/O stream is linked to a physical device by the Java I/O system. All I/O streams behave in the same manner, even if the actual physical devices they are linked to differ.

NOTE The stream-based I/O system packaged in **java.io** and described in this chapter has been part of Java since its original release and is widely used. However, beginning with version 1.4, a second I/O system was added to Java. It is called NIO (which was originally an acronym for New I/O). NIO is packaged in **java.nio** and its subpackages. The NIO system is described in Chapter 22.

NOTE It is important not to confuse the I/O streams used by the I/O system discussed here with the stream API added by JDK 8. Although conceptually related, they are two different things. Therefore, when the term *stream* is used in this chapter, it refers to an I/O stream.

The I/O Classes and Interfaces

The I/O classes defined by **java.io** are listed here:

BufferedInputStream	FileWriter	PipedInputStream
BufferedOutputStream	FilterInputStream	PipedOutputStream
BufferedReader	FilterOutputStream	PipedReader
BufferedWriter	FilterReader	PipedWriter
ByteArrayInputStream	FilterWriter	PrintStream
ByteArrayOutputStream	InputStream	PrintWriter
CharArrayReader	InputStreamReader	PushbackInputStream
CharArrayWriter	LineNumberReader	PushbackReader
Console	ObjectInputFilter.Config	RandomAccessFile
DataInputStream	ObjectInputStream	Reader
DataOutputStream	ObjectInputStream.GetField	SequenceInputStream
File	ObjectOutputStream	SerializablePermission
FileDescriptor	ObjectOutputStream.PutField	StreamTokenizer
FileInputStream	ObjectStreamClass	StringReader
FileOutputStream	ObjectStreamField	StringWriter
FilePermission	OutputStream	Writer
FileReader	OutputStreamWriter	

The **java.io** package also contains two deprecated classes that are not shown in the preceding table: **LineNumberInputStream** and **StringBufferInputStream**. These classes should not be used for new code.

The following interfaces are defined by **java.io**:

Closeable	FilenameFilter	ObjectInputValidation
DataInput	Flushable	ObjectOutput
DataOutput	ObjectInput	ObjectStreamConstants
Externalizable	ObjectInputFilter	Serializable
FileFilter	ObjectInputFilter.FilterInfo	

As you can see, there are many classes and interfaces in the **java.io** package. These include byte and character streams, and object serialization (the storage and retrieval of objects). This chapter examines several commonly used I/O components. We begin our discussion with one of the most distinctive I/O classes: **File**.

File

Although most of the classes defined by **java.io** operate on streams, the **File** class does not. It deals directly with files and the file system. That is, the **File** class does not specify how information is retrieved from or stored in files; it describes the properties of a file itself. A **File** object is used to obtain or manipulate the information associated with a disk file, such as the permissions, time, date, and directory path, and to navigate subdirectory hierarchies.

NOTE The **Path** interface and **Files** class, which are part of the NIO system, offer a powerful alternative to **File** in many cases. See Chapter 22 for details.

Files are a primary source and destination for data within many programs. Although there are severe restrictions on their use within untrusted code for security reasons, files are still a central resource for storing persistent and shared information. A directory in Java is treated simply as a **File** with one additional property—a list of filenames that can be examined by the **list( )** method.

The following constructors can be used to create **File** objects:

File(String *directoryPath*)
File(String *directoryPath*, String *filename*)
File(File *dirObj*, String *filename*)
File(URI *uriObj*)

Here, *directoryPath* is the path name of the file; *filename* is the name of the file or subdirectory; *dirObj* is a **File** object that specifies a directory; and *uriObj* is a **URI** object that describes a file.

The following example creates three files: **f1**, **f2**, and **f3**. The first **File** object is constructed with a directory path as the only argument. The second includes two arguments—the path and the filename. The third includes the file path assigned to **f1** and a filename; **f3** refers to the same file as **f2**.

```
File f1 = new File("/");
File f2 = new File("/","autoexec.bat");
File f3 = new File(f1,"autoexec.bat");
```

NOTE Java does the right thing with path separators between UNIX and Windows conventions. If you use a forward slash (/) on a Windows version of Java, the path will still resolve correctly. Remember, if you are using the Windows convention of a backslash character (\), you will need to use its escape sequence (\\) within a string.

File defines many methods that obtain the standard properties of a **File** object.

For example, **getName( )** returns the name of the file; **getParent( )** returns the name of the parent directory; and **exists( )** returns **true** if the file exists, **false** if it does not. The following example demonstrates several of the **File** methods. It assumes that a directory called **java** exists off the root directory and that it contains a file called **COPYRIGHT**.

```
// Demonstrate File.
import java.io.File;

class FileDemo {
  static void p(String s) {
    System.out.println(s);
  }

  public static void main(String args[]) {
    File f1 = new File("/java/COPYRIGHT");

    p("File Name: " + f1.getName());
    p("Path: " + f1.getPath());
    p("Abs Path: " + f1.getAbsolutePath());
    p("Parent: " + f1.getParent());
    p(f1.exists() ? "exists" : "does not exist");
    p(f1.canWrite() ? "is writeable" : "is not writeable");
    p(f1.canRead() ? "is readable" : "is not readable");
    p("is " + (f1.isDirectory() ? "" : "not" + " a directory"));
    p(f1.isFile() ? "is normal file" : "might be a named pipe");
    p(f1.isAbsolute() ? "is absolute" : "is not absolute");
    p("File last modified: " + f1.lastModified());
    p("File size: " + f1.length() + " Bytes");
  }
}
```

This program will produce output similar to this:

```
File Name: COPYRIGHT
Path: \java\COPYRIGHT
Abs Path: C:\java\COPYRIGHT
Parent: \java
exists
is writeable
is readable
is not a directory
is normal file
is not absolute
```

```
File last modified: 1282832030047
File size: 695 Bytes
```

Most of the **File** methods are self-explanatory. **isFile( )** and **isAbsolute( )** are not. **isFile( )** returns **true** if called on a file and **false** if called on a directory. Also, **isFile( )** returns **false** for some special files, such as device drivers and named pipes, so this method can be used to make sure the file will behave as a file. The **isAbsolute( )** method returns **true** if the file has an absolute path and **false** if its path is relative.

File includes two useful utility methods of special interest. The first is **renameTo( )**, shown here:

boolean renameTo(File *newName*)

Here, the filename specified by *newName* becomes the new name of the invoking **File** object. It will return **true** upon success and **false** if the file cannot be renamed (if you attempt to rename a file so that it uses an existing filename, for example).

The second utility method is **delete( )**, which deletes the disk file represented by the path of the invoking **File** object. It is shown here:

boolean delete( )

You can also use **delete( )** to delete a directory if the directory is empty. **delete( )** returns **true** if it deletes the file and **false** if the file cannot be removed.

Here are some other **File** methods that you will find helpful:

Method	Description
void deleteOnExit( )	Removes the file associated with the invoking object when the Java Virtual Machine terminates.
long getFreeSpace( )	Returns the number of free bytes of storage available on the partition associated with the invoking object.
long getTotalSpace( )	Returns the storage capacity of the partition associated with the invoking object.
long getUsableSpace( )	Returns the number of usable free bytes of storage available on the partition associated with the invoking object.
boolean isHidden( )	Returns **true** if the invoking file is hidden. Returns **false** otherwise.
boolean setLastModified(long *millisec*)	Sets the time stamp on the invoking file to that specified by *millisec*, which is the number of milliseconds from January 1, 1970, Coordinated Universal Time (UTC).
boolean setReadOnly( )	Sets the invoking file to read-only.

Methods also exist to mark files as readable, writable, and executable. Because **File** implements the **Comparable** interface, the method **compareTo( )** is also supported.

A method of special interest is called **toPath( )**, which is shown here:

```
Path toPath( )
```

toPath( ) returns a **Path** object that represents the file encapsulated by the invoking **File** object. (In other words, **toPath( )** converts a **File** into a **Path**.) **Path** is packaged in **java.nio.file** and is part of NIO. Thus, **toPath( )** forms a bridge between the older **File** class and the newer **Path** interface. (See Chapter 22 for a discussion of **Path**.)

Directories

A directory is a **File** that contains a list of other files and directories. When you create a **File** object that is a directory, the **isDirectory( )** method will return **true**. In this case, you can call **list( )** on that object to extract the list of other files and directories inside. It has two forms. The first is shown here:

```
String[ ] list( )
```

The list of files is returned in an array of **String** objects.

The program shown here illustrates how to use **list( )** to examine the contents

of a directory:

```
// Using directories.
import java.io.File;

class DirList {
  public static void main(String args[]) {
    String dirname = "/java";
    File f1 = new File(dirname);

    if (f1.isDirectory()) {
      System.out.println("Directory of " + dirname);
      String s[] = f1.list();

      for (int i=0; i < s.length; i++) {
        File f = new File(dirname + "/" + s[i]);
        if (f.isDirectory()) {
          System.out.println(s[i] + " is a directory");
        } else {
          System.out.println(s[i] + " is a file");
        }
      }
    } else {
      System.out.println(dirname + " is not a directory");
    }
  }
}
```

Here is sample output from the program. (Of course, the output you see will be different, based on what is in the directory.)

```
Directory of /java
bin is a directory
lib is a directory
demo is a directory
COPYRIGHT is a file
README is a file
index.html is a file
include is a directory
src.zip is a file
src is a directory
```

Using FilenameFilter

You will often want to limit the number of files returned by the **list( )** method to include only those files that match a certain filename pattern, or *filter*. To do this, you must use a second form of **list( )**, shown here:

String[ ] list(FilenameFilter *FFObj*)

In this form, *FFObj* is an object of a class that implements the **FilenameFilter** interface.

FilenameFilter defines only a single method, **accept( )**, which is called once for each file in a list. Its general form is given here:

boolean accept(File *directory*, String *filename*)

The **accept( )** method returns **true** for files in the directory specified by *directory* that should be included in the list (that is, those that match the *filename* argument) and returns **false** for those files that should be excluded.

The **OnlyExt** class, shown next, implements **FilenameFilter**. It will be used to modify the preceding program to restrict the visibility of the filenames returned by **list( )** to files with names that end in the file extension specified when the object is constructed.

```
import java.io.*;

public class OnlyExt implements FilenameFilter {
  String ext;

  public OnlyExt(String ext) {
    this.ext = "." + ext;
  }

  public boolean accept(File dir, String name) {
    return name.endsWith(ext);
  }
}
```

The modified directory listing program is shown here. Now it will only display files that use the **.html** extension.

```
// Directory of .HTML files.
import java.io.*;

class DirListOnly {
  public static void main(String args[]) {
    String dirname = "/java";
    File f1 = new File(dirname);
    FilenameFilter only = new OnlyExt("html");
    String s[] = f1.list(only);

    for (int i=0; i < s.length; i++) {
      System.out.println(s[i]);
    }
  }
}
```

The listFiles( ) Alternative

There is a variation to the **list( )** method, called **listFiles( )**, which you might find useful. The signatures for **listFiles( )** are shown here:

File[ ] listFiles( )
File[ ] listFiles(FilenameFilter *FFObj*)
File[ ] listFiles(FileFilter *FObj*)

These methods return the file list as an array of **File** objects instead of strings. The first method returns all files, and the second returns those files that satisfy the specified **FilenameFilter**. Aside from returning an array of **File** objects, these two versions of **listFiles( )** work like their equivalent **list( )** methods.

The third version of **listFiles( )** returns those files with path names that satisfy the specified **FileFilter**. **FileFilter** defines only a single method, **accept( )**, which is called once for each file in a list. Its general form is given here:

boolean accept(File *path*)

The **accept( )** method returns **true** for files that should be included in the list (that is, those that match the *path* argument) and **false** for those that should be excluded.

Creating Directories

Another two useful **File** utility methods are **mkdir( )** and **mkdirs( )**. The **mkdir(**

) method creates a directory, returning **true** on success and **false** on failure. Failure can occur for various reasons, such as the path specified in the **File** object already exists, or the directory cannot be created because the entire path does not exist yet. To create a directory for which no path exists, use the **mkdirs( )** method. It creates both a directory and all the parents of the directory.

The AutoCloseable, Closeable, and Flushable Interfaces

There are three interfaces that are quite important to the stream classes. Two are **Closeable** and **Flushable**. They are defined in **java.io**. The third, **AutoCloseable**, is packaged in **java.lang**.

AutoCloseable provides support for the **try**-with-resources statement, which automates the process of closing a resource. (See Chapter 13.) Only objects of classes that implement **AutoCloseable** can be managed by **try**-with-resources. **AutoCloseable** is discussed in Chapter 17, but it is reviewed here for convenience. The **AutoCloseable** interface defines only the **close( )** method:

```
void close( ) throws Exception
```

This method closes the invoking object, releasing any resources that it may hold. It is called automatically at the end of a **try**-with-resources statement, thus eliminating the need to explicitly call **close( )**. Because this interface is implemented by all of the I/O classes that open a stream, all such streams can be automatically closed by a **try**-with-resources statement. Automatically closing a stream ensures that it is properly closed when it is no longer needed, thus preventing memory leaks and other problems.

The **Closeable** interface also defines the **close( )** method. Objects of a class that implement **Closeable** can be closed. **Closeable** extends **AutoCloseable**. Therefore, any class that implements **Closeable** also implements **AutoCloseable**.

Objects of a class that implements **Flushable** can force buffered output to be written to the stream to which the object is attached. It defines the **flush( )** method, shown here:

```
void flush( ) throws IOException
```

Flushing a stream typically causes buffered output to be physically written to the underlying device. This interface is implemented by all of the I/O classes that

write to a stream.

I/O Exceptions

Two exceptions play an important role in I/O handling. The first is **IOException**. As it relates to most of the I/O classes described in this chapter, if an I/O error occurs, an **IOException** is thrown. In many cases, if a file cannot be opened, a **FileNotFoundException** is thrown. **FileNotFoundException** is a subclass of **IOException**, so both can be caught with a single **catch** that catches **IOException**. For brevity, this is the approach used by most of the sample code in this chapter. However, in your own applications, you might find it useful to **catch** each exception separately.

Another exception class that is sometimes important when performing I/O is **SecurityException**. As explained in Chapter 13, in situations in which a security manager is present, several of the file classes will throw a **SecurityException** if a security violation occurs when attempting to open a file. By default, applications run via **java** do not use a security manager. For that reason, the I/O examples in this book do not need to watch for a possible **SecurityException**. However, other applications could generate a **SecurityException**. In such a case, you will need to handle this exception.

Two Ways to Close a Stream

In general, a stream must be closed when it is no longer needed. Failure to do so can lead to memory leaks and resource starvation. The techniques used to close a stream were described in Chapter 13, but because of their importance, they warrant a brief review here before the stream classes are examined.

Beginning with JDK 7, there are two basic ways in which you can close a stream. The first is to explicitly call **close( )** on the stream. This is the traditional approach that has been used since the original release of Java. With this approach, **close( )** is typically called within a **finally** block. Thus, a simplified skeleton for the traditional approach is shown here:

```
try {
// open and access file
} catch( I/O-exception) {
// ...
} finally {
```

```
// close the file
}
```

This general technique (or variation thereof) is common in code that predates JDK 7.

The second approach to closing a stream is to automate the process by using the **try**-with-resources statement that was added by JDK 7. The **try**-with-resources statement is an enhanced form of **try** that has the following form:

```
try (resource-specification) {
// use the resource
}
```

Typically, *resource-specification* is a statement or statements that declares and initializes a resource, such as a file or other stream-related resource. It consists of a variable declaration in which the variable is initialized with a reference to the object being managed. When the **try** block ends, the resource is automatically released. In the case of a file, this means that the file is automatically closed. Thus, there is no need to call **close( )** explicitly. Beginning with JDK 9, it is also possible for the resource specification of the **try** to consist of a variable that has been declared and initialized earlier in the program. However, that variable must be effectively **final**, which means that it has not been assigned a new value after being given its initial value.

Here are three key points about the **try**-with-resources statement:

- Resources managed by **try**-with-resources must be objects of classes that implement **AutoCloseable**.
- A resource declared in the **try** is implicitly **final**. A resource declared outside the **try** must be effectively **final**.
- You can manage more than one resource by separating each declaration by a semicolon.

Also, remember that the scope of a resource declared inside the **try** is limited to the **try**-with-resources statement.

The principal advantage of **try**-with-resources is that the resource (in this case, a stream) is closed automatically when the **try** block ends. Thus, it is not possible to forget to close the stream, for example. The **try**-with-resources approach also typically results in shorter, clearer, easier-to-maintain source code.

Because of its advantages, **try**-with-resources is expected to be used extensively in new code. As a result, most of the code in this chapter (and in this

book) will use it. However, because a large amount of older code still exists, it is important for all programmers to also be familiar with the traditional approach to closing a stream. For example, you will quite likely have to work on legacy code that uses the traditional approach or in an environment that uses an older version of Java. There may also be times when the automated approach is not appropriate because of other aspects of your code. For this reason, a few I/O examples in this book will demonstrate the traditional approach so you can see it in action.

One last point: The examples that use **try**-with-resources must be compiled by a modern version of Java. They won't work with an older compiler. The examples that use the traditional approach can be compiled by older versions of Java.

REMEMBER Because **try**-with-resources streamlines the process of releasing a resource and eliminates the possibility of accidentally forgetting to release a resource, it is the approach recommended for new code when its use is appropriate.

The Stream Classes

Java's stream-based I/O is built upon four abstract classes: **InputStream**, **OutputStream**, **Reader**, and **Writer**. These classes were briefly discussed in Chapter 13. They are used to create several concrete stream subclasses. Although your programs perform their I/O operations through concrete subclasses, the top-level classes define the basic functionality common to all stream classes.

InputStream and **OutputStream** are designed for byte streams. **Reader** and **Writer** are designed for character streams. The byte stream classes and the character stream classes form separate hierarchies. In general, you should use the character stream classes when working with characters or strings and use the byte stream classes when working with bytes or other binary objects.

In the remainder of this chapter, both the byte- and character-oriented streams are examined.

The Byte Streams

The byte stream classes provide a rich environment for handling byte-oriented I/O. A byte stream can be used with any type of object, including binary data. This versatility makes byte streams important to many types of programs. Since

the byte stream classes are topped by **InputStream** and **OutputStream**, our discussion begins with them.

InputStream

InputStream is an abstract class that defines Java's model of streaming byte input. It implements the **AutoCloseable** and **Closeable** interfaces. Most of the methods in this class will throw an **IOException** when an I/O error occurs. (The exceptions are **mark( )** and **markSupported( )**.) Table 21-1 shows the methods in **InputStream**.

Method	Description
int available( )	Returns the number of bytes of input currently available for reading.
void close( )	Closes the input source. Further read attempts will generate an **IOException**.
void mark(int *numBytes*)	Places a mark at the current point in the input stream that will remain valid until *numBytes* bytes are read.
boolean markSupported( )	Returns **true** if **mark( )** / **reset( )** are supported by the invoking stream.
static InputStream nullInputStream( )	Returns an open, but null input stream, which is a stream that contains no data. Thus, the stream is always at the end of the stream and no input can be obtained. The stream can, however, be closed. (Added by JDK 11.)
int read( )	Returns an integer representation of the next available byte of input. –1 is returned when an attempt is made to read at the end of the stream.
int read(byte *buffer*[ ])	Attempts to read up to *buffer.length* bytes into *buffer* and returns the actual number of bytes that were successfully read. –1 is returned when an attempt is made to read at the end of the stream.
int read(byte *buffer*[ ], int *offset*, int *numBytes*)	Attempts to read up to *numBytes* bytes into *buffer* starting at *buffer*[*offset*], returning the number of bytes successfully read. –1 is returned when an attempt is made to read at the end of the stream.
byte[ ] readAllBytes( )	Beginning at the current position, reads to the end of the stream, returning a byte array that holds the input.
byte[ ] readNBytes(int *numBytes*)	Attempts to read *numBytes* bytes, returning the result in a byte array. If the end of the stream is reached before *numBytes* bytes have been read, then the returned array will contain less than *numBytes* bytes. (Added by JDK 11.)
int readNBytes(byte *buffer*[ ], int *offset*, int *numBytes*)	Attempts to read up to *numBytes* bytes into *buffer* starting at *buffer*[*offset*], returning the number of bytes successfully read.
void reset( )	Resets the input pointer to the previously set mark.
long skip(long *numBytes*)	Ignores (that is, skips) *numBytes* bytes of input, returning the number of bytes actually ignored.
long transferTo(OutputStream *strm*)	Copies the bytes in the invoking stream into *strm*, returning the number of bytes copied.

Table 21-1 The Methods Defined by **InputStream**

NOTE Most of the methods described in Table 21-1 are implemented by the subclasses of **InputStream**. The **mark( )** and **reset( )** methods are exceptions; notice their use, or lack thereof, by each subclass in the discussions that follow.

OutputStream

OutputStream is an abstract class that defines streaming byte output. It implements the **AutoCloseable**, **Closeable**, and **Flushable** interfaces. Most of the methods defined by this class return **void** and throw an **IOException** in the case of I/O errors. Table 21-2 shows the methods in **OutputStream**.

Method	Description
void close( )	Closes the output stream. Further write attempts will generate an **IOException**.
void flush( )	Finalizes the output state so that any buffers are cleared. That is, it flushes the output buffers.
static OutputStream nullOutputStream( )	Returns an open, but null output stream, which is a stream to which no output is actually written. Thus, its output methods can be called but don't actually produce output. The stream can, however, be closed. (Added by JDK 11.)
void write(int *b*)	Writes a single byte to an output stream. Note that the parameter is an **int**, which allows you to call **write( )** with an expression without having to cast it back to **byte**.
void write(byte *buffer*[ ])	Writes a complete array of bytes to an output stream.
void write(byte *buffer*[ ], int *offset*, int *numBytes*)	Writes a subrange of *numBytes* bytes from the array *buffer*, beginning at *buffer*[*offset*].

Table 21-2 The Methods Defined by **OutputStream**

FileInputStream

The **FileInputStream** class creates an **InputStream** that you can use to read bytes from a file. Two commonly used constructors are shown here:

FileInputStream(String *filePath*)
FileInputStream(File *fileObj*)

Either can throw a **FileNotFoundException**. Here, *filePath* is the full path name of a file, and *fileObj* is a **File** object that describes the file.

The following example creates two **FileInputStream**s that use the same disk file and each of the two constructors:

```
FileInputStream f0 = new FileInputStream("/autoexec.bat")
File f = new File("/autoexec.bat");
```

```
FileInputStream f1 = new FileInputStream(f);
```

Although the first constructor is probably more commonly used, the second allows you to closely examine the file using the **File** methods, before attaching it to an input stream. When a **FileInputStream** is created, it is also opened for reading. **FileInputStream** overrides several of the methods in the abstract class **InputStream**. The **mark( )** and **reset( )** methods are not overridden, and any attempt to use **reset( )** on a **FileInputStream** will generate an **IOException**.

The next example shows how to read a single byte, an array of bytes, and a subrange of an array of bytes. It also illustrates how to use **available( )** to determine the number of bytes remaining and how to use the **skip( )** method to skip over unwanted bytes. The program reads its own source file, which must be in the current directory. Notice that it uses the **try**-with-resources statement to automatically close the file when it is no longer needed.

```
// Demonstrate FileInputStream.

import java.io.*;

class FileInputStreamDemo {
  public static void main(String args[]) {
    int size;

    // Use try-with-resources to close the stream.
    try ( FileInputStream f =
           new FileInputStream("FileInputStreamDemo.java") ) {

      System.out.println("Total Available Bytes: " +
                         (size = f.available()));

      int n = size/40;
      System.out.println("First " + n +
                         " bytes of the file one read() at a time");
      for (int i=0; i < n; i++) {
        System.out.print((char) f.read());
      }

      System.out.println("\nStill Available: " + f.available());

      System.out.println("Reading the next " + n +
                         " with one read(b[])");
      byte b[] = new byte[n];
      if (f.read(b) != n) {
        System.err.println("couldn't read " + n + " bytes.");
      }

      System.out.println(new String(b, 0, n));
      System.out.println("\nStill Available: " + (size = f.available()));
      System.out.println("Skipping half of remaining bytes with skip()");
      f.skip(size/2);
      System.out.println("Still Available: " + f.available());

      System.out.println("Reading " + n/2 + " into the end of array");
      if (f.read(b, n/2, n/2) != n/2) {
        System.err.println("couldn't read " + n/2 + " bytes.");
      }

      System.out.println(new String(b, 0, b.length));
      System.out.println("\nStill Available: " + f.available());
    } catch(IOException e) {
      System.out.println("I/O Error: " + e);
    }
  }
}
```

Here is the output produced by this program:

```
Total Available Bytes: 1714
First 42 bytes of the file one read() at a time
// Demonstrate FileInputStream.

impor
Still Available: 1672
Reading the next 42 with one read(b[])
t java.io.*;

class FileInputStreamD

Still Available: 1630
Skipping half of remaining bytes with skip()
Still Available: 815
Reading 21 into the end of array
t java.io.*;

c n) {
        Syst

Still Available: 794
```

This somewhat contrived example demonstrates how to read three ways, to skip input, and to inspect the amount of data available on a stream.

NOTE The preceding example and the other examples in this chapter handle any I/O exceptions that might occur as described in Chapter 13. See Chapter 13 for details and alternatives.

FileOutputStream

FileOutputStream creates an **OutputStream** that you can use to write bytes to a file. It implements the **AutoCloseable**, **Closeable**, and **Flushable** interfaces. Four of its constructors are shown here:

FileOutputStream(String *filePath*)
FileOutputStream(File *fileObj*)
FileOutputStream(String *filePath*, boolean *append*)
FileOutputStream(File *fileObj*, boolean *append*)

They can throw a **FileNotFoundException**. Here, *filePath* is the full path name

of a file, and *fileObj* is a **File** object that describes the file. If *append* is **true**, the file is opened in append mode.

Creation of a **FileOutputStream** is not dependent on the file already existing. **FileOutputStream** will create the file before opening it for output when you create the object. In the case where you attempt to open a read-only file, an exception will be thrown.

The following example creates a sample buffer of bytes by first making a **String** and then using the **getBytes( )** method to extract the byte array equivalent. It then creates three files. The first, **file1.txt**, will contain every other byte from the sample. The second, **file2.txt**, will contain the entire set of bytes. The third and last, **file3.txt**, will contain only the last quarter.

```
// Demonstrate FileOutputStream.
// This program uses the traditional approach to closing a file.

import java.io.*;

class FileOutputStreamDemo {
  public static void main(String args[]) {
    String source = "Now is the time for all good men\n"
      + " to come to the aid of their country\n"
      + " and pay their due taxes.";
    byte buf[] = source.getBytes();
    FileOutputStream f0 = null;
    FileOutputStream f1 = null;
    FileOutputStream f2 = null;

    try {
      f0 = new FileOutputStream("file1.txt");
      f1 = new FileOutputStream("file2.txt");
      f2 = new FileOutputStream("file3.txt");

      // write to first file
      for (int i=0; i < buf.length; i += 2) f0.write(buf[i]);

      // write to second file
      f1.write(buf);

      // write to third file
      f2.write(buf, buf.length-buf.length/4, buf.length/4);
    } catch(IOException e) {
      System.out.println("An I/O Error Occurred");
    } finally {
      try {
        if(f0 != null) f0.close();
      } catch(IOException e) {
        System.out.println("Error Closing file1.txt");
      }
      try {
        if(f1 != null) f1.close();
      } catch(IOException e) {
        System.out.println("Error Closing file2.txt");
      }
      try {
        if(f2 != null) f2.close();
      } catch(IOException e) {
        System.out.println("Error Closing file3.txt");
      }
    }
  }
}
```

Here are the contents of each file after running this program. First, **file1.txt**:

```
Nwi h iefralgo e
t oet h i ftercuty n a hi u ae.
```

Next, **file2.txt**:

```
Now is the time for all good men
 to come to the aid of their country
 and pay their due taxes.
```

Finally, **file3.txt**:

```
nd pay their due taxes.
```

As the comment at the top of the program states, the preceding program shows an example that uses the traditional approach to closing a file when it is no longer needed. This approach is required by all versions of Java prior to JDK 7 and is widely used in legacy code. As you can see, quite a bit of rather awkward code is required to explicitly call **close( )** because each call could generate an **IOException** if the close operation fails. This program can be substantially improved by using the **try**-with-resources statement. For comparison, here is the revised version. Notice that it is much shorter and streamlined:

```
// Demonstrate FileOutputStream.
// This version uses try-with-resources.

import java.io.*;

class FileOutputStreamDemo {
  public static void main(String args[]) {
    String source = "Now is the time for all good men\n"
      + " to come to the aid of their country\n"
      + " and pay their due taxes.";
    byte buf[] = source.getBytes();

    // Use try-with-resources to close the files.
    try (FileOutputStream f0 = new FileOutputStream("file1.txt");
         FileOutputStream f1 = new FileOutputStream("file2.txt");
         FileOutputStream f2 = new FileOutputStream("file3.txt") )
    {

      // write to first file
      for (int i=0; i < buf.length; i += 2) f0.write(buf[i]);

      // write to second file
      f1.write(buf);

      // write to third file
      f2.write(buf, buf.length-buf.length/4, buf.length/4);
    } catch(IOException e) {
      System.out.println("An I/O Error Occurred");
    }
  }
}
```

ByteArrayInputStream

ByteArrayInputStream is an implementation of an input stream that uses a byte array as the source. This class has two constructors, each of which requires a byte array to provide the data source:

ByteArrayInputStream(byte *array* [ ])
ByteArrayInputStream(byte *array* [ ], int *start*, int *numBytes*)

Here, *array* is the input source. The second constructor creates an **InputStream** from a subset of the byte array that begins with the character at the index

specified by *start* and is *numBytes* long.

The **close( )** method has no effect on a **ByteArrayInputStream**. Therefore, it is not necessary to call **close( )** on a **ByteArrayInputStream**, but doing so is not an error.

The following example creates a pair of **ByteArrayInputStream**s, initializing them with the byte representation of the alphabet:

```
// Demonstrate ByteArrayInputStream.
import java.io.*;

class ByteArrayInputStreamDemo {
  public static void main(String args[]) {
    String tmp = "abcdefghijklmnopqrstuvwxyz";
    byte b[] = tmp.getBytes();

    ByteArrayInputStream input1 = new ByteArrayInputStream(b);
    ByteArrayInputStream input2 = new ByteArrayInputStream(b,0,3);
  }
}
```

The **input1** object contains the entire lowercase alphabet, whereas **input2** contains only the first three letters.

A **ByteArrayInputStream** implements both **mark( )** and **reset( )**. However, if **mark( )** has not been called, then **reset( )** sets the stream pointer to the start of the stream—which, in this case, is the start of the byte array passed to the constructor. The next example shows how to use the **reset( )** method to read the same input twice. In this case, the program reads and prints the letters "abc" once in lowercase and then again in uppercase.

```
import java.io.*;

class ByteArrayInputStreamReset {
  public static void main(String args[]) {
    String tmp = "abc";
    byte b[] = tmp.getBytes();
    ByteArrayInputStream in = new ByteArrayInputStream(b);

    for (int i=0; i<2; i++) {
      int c;
      while ((c = in.read()) != -1) {
        if (i == 0) {
          System.out.print((char) c);
        } else {
          System.out.print(Character.toUpperCase((char) c));
        }
      }
      System.out.println();
      in.reset();
    }
  }
}
```

This example first reads each character from the stream and prints it as-is in lowercase. It then resets the stream and begins reading again, this time converting each character to uppercase before printing. Here's the output:

```
abc
ABC
```

ByteArrayOutputStream

ByteArrayOutputStream is an implementation of an output stream that uses a byte array as the destination. **ByteArrayOutputStream** has two constructors, shown here:

ByteArrayOutputStream( )
ByteArrayOutputStream(int *numBytes*)

In the first form, a buffer of 32 bytes is created. In the second, a buffer is created with a size equal to that specified by *numBytes*. The buffer is held in the protected **buf** field of **ByteArrayOutputStream**. The buffer size will be

increased automatically, if needed. The number of bytes held by the buffer is contained in the protected **count** field of **ByteArrayOutputStream**.

The **close( )** method has no effect on a **ByteArrayOutputStream**. Therefore, it is not necessary to call **close( )** on a **ByteArrayOutputStream**, but doing so is not an error.

The following example demonstrates **ByteArrayOutputStream**:

```
// Demonstrate ByteArrayOutputStream.

import java.io.*;

class ByteArrayOutputStreamDemo {
  public static void main(String args[]) {
    ByteArrayOutputStream f = new ByteArrayOutputStream();
    String s = "This should end up in the array";
    byte buf[] = s.getBytes();

    try {
      f.write(buf);
    } catch(IOException e) {
      System.out.println("Error Writing to Buffer");
      return;
    }

    System.out.println("Buffer as a string");
    System.out.println(f.toString());
    System.out.println("Into array");
    byte b[] = f.toByteArray();
    for (int i=0; i<b.length; i++) System.out.print((char) b[i]);

    System.out.println("\nTo an OutputStream()");

    // Use try-with-resources to manage the file stream.
    try ( FileOutputStream f2 = new FileOutputStream("test.txt") )
    {
      f.writeTo(f2);
    } catch(IOException e) {
      System.out.println("I/O Error: " + e);
      return;
    }

    System.out.println("Doing a reset");
    f.reset();

    for (int i=0; i\<3; i++) f.write('X');

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

When you run the program, you will create the following output. Notice how after the call to **reset( )**, the three X's end up at the beginning.

```
Buffer as a string
This should end up in the array
Into array
This should end up in the array
To an OutputStream()
Doing a reset
XXX
```

This example uses the **writeTo( )** convenience method to write the contents of **f** to **test.txt**. Examining the contents of the **test.txt** file created in the preceding example shows the result we expected:

```
This should end up in the array
```

Filtered Byte Streams

Filtered streams are simply wrappers around underlying input or output streams that transparently provide some extended level of functionality. These streams are typically accessed by methods that are expecting a generic stream, which is a superclass of the filtered streams. Typical extensions are buffering, character translation, and raw data translation. The filtered byte streams are **FilterInputStream** and **FilterOutputStream**. Their constructors are shown here:

FilterOutputStream(OutputStream *os*)
FilterInputStream(InputStream *is*)

The methods provided in these classes are identical to those in **InputStream** and **OutputStream**.

Buffered Byte Streams

For the byte-oriented streams, a *buffered stream* extends a filtered stream class by attaching a memory buffer to the I/O stream. This buffer allows Java to do I/O operations on more than a byte at a time, thereby improving performance. Because the buffer is available, skipping, marking, and resetting of the stream become possible. The buffered byte stream classes are **BufferedInputStream** and **BufferedOutputStream**. **PushbackInputStream** also implements a

buffered stream.

BufferedInputStream

Buffering I/O is a very common performance optimization. Java's **BufferedInputStream** class allows you to "wrap" any **InputStream** into a buffered stream to improve performance.

BufferedInputStream has two constructors:

BufferedInputStream(InputStream *inputStream*)
BufferedInputStream(InputStream *inputStream*, int *bufSize*)

The first form creates a buffered stream using a default buffer size. In the second, the size of the buffer is passed in *bufSize*. Use of sizes that are multiples of a memory page, a disk block, and so on, can have a significant positive impact on performance. This is, however, implementation-dependent. An optimal buffer size is generally dependent on the host operating system, the amount of memory available, and how the machine is configured. To make good use of buffering doesn't necessarily require quite this degree of sophistication. A good guess for a size is around 8,192 bytes, and attaching even a rather small buffer to an I/O stream is always a good idea. That way, the low-level system can read blocks of data from the disk or network and store the results in your buffer. Thus, even if you are reading the data a byte at a time out of the **InputStream**, you will be manipulating fast memory most of the time.

Buffering an input stream also provides the foundation required to support moving backward in the stream of the available buffer. Beyond the **read( )** and **skip( )** methods implemented in any **InputStream**, **BufferedInputStream** also supports the **mark( )** and **reset( )** methods. This support is reflected by **BufferedInputStream.markSupported( )** returning **true**.

The following example contrives a situation where we can use **mark( )** to remember where we are in an input stream and later use **reset( )** to get back there. This example is parsing a stream for the HTML entity reference for the copyright symbol. Such a reference begins with an ampersand (&) and ends with a semicolon (;) without any intervening whitespace. The sample input has two ampersands to show the case where the **reset( )** happens and where it does not.

```
// Use buffered input.

import java.io.*;

class BufferedInputStreamDemo {
  public static void main(String args[]) {
    String s = "This is a &copy; copyright symbol " +
      "but this is &copy not.\n";
    byte buf[] = s.getBytes();

    ByteArrayInputStream in = new ByteArrayInputStream(buf);
    int c;
    boolean marked = false;

    // Use try-with-resources to manage the file.
    try ( BufferedInputStream f = new BufferedInputStream(in) )
    {
      while ((c = f.read()) != -1) {
        switch(c) {
        case '&':
          if (!marked) {
            f.mark(32);
            marked = true;
          } else {
            marked = false;
          }
          break;
```

```
          case ';':
            if (marked) {
              marked = false;
              System.out.print("(c)");
            } else
              System.out.print((char) c);
            break;
          case ' ':
            if (marked) {
              marked = false;
              f.reset();
              System.out.print("&");
            } else
              System.out.print((char) c);
            break;
          default:
            if (!marked)
              System.out.print((char) c);
            break;
          }
        }
      } catch(IOException e) {
        System.out.println("I/O Error: " + e);
      }
    }
}
```

Notice that this example uses **mark(32)**, which preserves the mark for the next 32 bytes read (which is enough for all entity references). Here is the output produced by this program:

```
This is a (c) copyright symbol but this is &copy not.
```

BufferedOutputStream

A **BufferedOutputStream** is similar to any **OutputStream** with the exception that the **flush( )** method is used to ensure that data buffers are written to the stream being buffered. Since the point of a **BufferedOutputStream** is to improve performance by reducing the number of times the system actually writes data, you may need to call **flush( )** to cause any data that is in the buffer to be immediately written.

Unlike buffered input, buffering output does not provide additional

functionality. Buffers for output in Java are there to increase performance. Here are the two available constructors:

BufferedOutputStream(OutputStream *outputStream*)
BufferedOutputStream(OutputStream *outputStream*, int *bufSize*)

The first form creates a buffered stream using the default buffer size. In the second form, the size of the buffer is passed in *bufSize*.

PushbackInputStream

One of the novel uses of buffering is the implementation of pushback. *Pushback* is used on an input stream to allow a byte to be read and then returned (that is, "pushed back") to the stream. The **PushbackInputStream** class implements this idea. It provides a mechanism to "peek" at what is coming from an input stream without disrupting it.

PushbackInputStream has the following constructors:

PushbackInputStream(InputStream *inputStream*)
PushbackInputStream(InputStream *inputStream*, int *numBytes*)

The first form creates a stream object that allows one byte to be returned to the input stream. The second form creates a stream that has a pushback buffer that is *numBytes* long. This allows multiple bytes to be returned to the input stream.

Beyond the familiar methods of **InputStream**, **PushbackInputStream** provides **unread( )**, shown here:

void unread(int *b*)
void unread(byte *buffer* [ ])
void unread(byte *buffer*, int *offset*, int *numBytes*)

The first form pushes back the low-order byte of *b*. This will be the next byte returned by a subsequent call to **read( )**. The second form pushes back the bytes in *buffer*. The third form pushes back *numBytes* bytes beginning at *offset* from *buffer*. An **IOException** will be thrown if there is an attempt to push back a byte when the pushback buffer is full.

Here is an example that shows how a programming language parser might use a **PushbackInputStream** and **unread( )** to deal with the difference between the = = operator for comparison and the = operator for assignment:

```
// Demonstrate unread().

import java.io.*;

class PushbackInputStreamDemo {
  public static void main(String args[]) {
    String s = "if (a == 4) a = 0;\n";
    byte buf[] = s.getBytes();
    ByteArrayInputStream in = new ByteArrayInputStream(buf);
    int c;

    try ( PushbackInputStream f = new PushbackInputStream(in) )
    {
      while ((c = f.read()) != -1) {
        switch(c) {
        case '=':
          if ((c = f.read()) == '=')
            System.out.print(".eq.");
          else {
            System.out.print("<-");
            f.unread(c);
          }
          break;
        default:
          System.out.print((char) c);
          break;
        }
      }
    } catch(IOException e) {
      System.out.println("I/O Error: " + e);
    }
  }
}
```

Here is the output for this example. Notice that == was replaced by **".eq."** and = was replaced by "<–".

```
if (a .eq. 4) a <- 0;
```

CAUTION PushbackInputStream has the side effect of invalidating the **mark( )** or **reset( )** methods of the **InputStream** used to create it. Use **markSupported( )** to check any stream on which you are going to use **mark( )/reset( )**.

SequenceInputStream

The **SequenceInputStream** class allows you to concatenate multiple **InputStream**s. The construction of a **SequenceInputStream** is different from any other **InputStream**. A **SequenceInputStream** constructor uses either a pair of **InputStream**s or an **Enumeration** of **InputStream**s as its argument:

SequenceInputStream(InputStream *first*, InputStream *second*)
SequenceInputStream(Enumeration <? extends InputStream> *streamEnum*)

Operationally, the class fulfills read requests from the first **InputStream** until it runs out and then switches over to the second one. In the case of an **Enumeration**, it will continue through all of the **InputStream**s until the end of the last one is reached. When the end of each file is reached, its associated stream is closed. Closing the stream created by **SequenceInputStream** causes all unclosed streams to be closed.

Here is a simple example that uses a **SequenceInputStream** to output the contents of two files. For demonstration purposes, this program uses the traditional technique used to close a file. As an exercise, you might want to try changing it to use the **try**-with-resources statement.

```
// Demonstrate sequenced input.
// This program uses the traditional approach to closing a file.

import java.io.*;
import java.util.*;

class InputStreamEnumerator implements Enumeration<FileInputStream> {
  private Enumeration<String> files;

  public InputStreamEnumerator(Vector<String> files) {
    this.files = files.elements();
  }

  public boolean hasMoreElements() {
    return files.hasMoreElements();
  }

  public FileInputStream nextElement() {
    try {
      return new FileInputStream(files.nextElement().toString());

    } catch (IOException e) {
      return null;
    }
  }
}
```

```
class SequenceInputStreamDemo {
  public static void main(String args[]) {
    int c;
    Vector<String> files = new Vector<String>();

    files.addElement("file1.txt");
    files.addElement("file2.txt");
    files.addElement("file3.txt");
    InputStreamEnumerator ise = new InputStreamEnumerator(files);
    InputStream input = new SequenceInputStream(ise);

    try {
      while ((c = input.read()) != -1)
        System.out.print((char) c);
    } catch(NullPointerException e) {
      System.out.println("Error Opening File.");
    } catch(IOException e) {
      System.out.println("I/O Error: " + e);
    } finally {
      try {
        input.close();
      } catch(IOException e) {
        System.out.println("Error Closing SequenceInputStream");
      }
    }
  }
}
```

This example creates a **Vector** and then adds three filenames to it. It passes that vector of names to the **InputStreamEnumerator** class, which is designed to provide a wrapper on the vector where the elements returned are not the filenames but, rather, open **FileInputStream**s on those names. The **SequenceInputStream** opens each file in turn, and this example prints the contents of the files.

Notice in **nextElement( )** that if a file cannot be opened, **null** is returned. This results in a **NullPointerException**, which is caught in **main( )**.

PrintStream

The **PrintStream** class provides all of the output capabilities we have been using from the **System** file handle, **System.out**, since the beginning of the book.

This makes **PrintStream** one of Java's most often used classes. It implements the **Appendable**, **AutoCloseable**, **Closeable**, and **Flushable** interfaces.

PrintStream defines several constructors. The ones shown next have been specified from the start:

PrintStream(OutputStream *outputStream*)
PrintStream (OutputStream *outputStream*, boolean *autoFlushingOn*)
PrintStream(OutputStream *outputStream*, boolean *autoFlushingOn* String *charSet*)
throws UnsupportedEncodingException

Here, *outputStream* specifies an open **OutputStream** that will receive output. The *autoFlushingOn* parameter controls whether the output buffer is automatically flushed every time a newline (**\n**) character or a byte array is written or when **println( )** is called. If *autoFlushingOn* is **true**, flushing automatically takes place. If it is **false**, flushing is not automatic. The first constructor does not automatically flush. You can specify a character encoding by passing its name in *charSet*.

The next set of constructors gives you an easy way to construct a **PrintStream** that writes its output to a file:

PrintStream(File *outputFile*) throws FileNotFoundException
PrintStream(File *outputFile*, String *charSet*)
throws FileNotFoundException, UnsupportedEncodingException
PrintStream(String *outputFileName*) throws FileNotFoundException
PrintStream(String *outputFileName*, String *charSet*) throws FileNotFoundException,
UnsupportedEncodingException

These allow a **PrintStream** to be created from a **File** object or by specifying the name of a file. In either case, the file is automatically created. Any preexisting file by the same name is destroyed. Once created, the **PrintStream** object directs all output to the specified file. You can specify a character encoding by passing its name in *charSet*. JDK 11 adds constructors that let you specify a **Charset** parameter.

NOTE If a security manager is present, some **PrintStream** constructors will throw a **SecurityException** if a security violation occurs.

PrintStream supports the **print( )** and **println( )** methods for all types,

including **Object**. If an argument is not a primitive type, the **PrintStream** methods will call the object's **toString( )** method and then display the result.

A number of years ago a very useful method called **printf( )** was added to **PrintStream**. It allows you to specify the precise format of the data to be written. The **printf( )** method formats as described by the **Formatter** class discussed in Chapter 20. It then writes this data to the invoking stream. Although formatting can be done manually, by using **Formatter** directly, **printf( )** streamlines the process. It also parallels the C/C++ **printf( )** function, which makes it easy to convert existing C/C++ code into Java. Frankly, **printf( )** was a much welcome addition to the Java API because it greatly simplified the output of formatted data to the console.

The **printf( )** method has the following general forms:

PrintStream printf(String *fmtString*, Object … *args*)
PrintStream printf(Locale *loc*, String *fmtString*, Object … *args*)

The first version writes *args* to standard output in the format specified by *fmtString*, using the default locale. The second lets you specify a locale. Both return the invoking **PrintStream**.

In general, **printf( )** works in a manner similar to the **format( )** method specified by **Formatter**. The *fmtString* consists of two types of items. The first type is composed of characters that are simply copied to the output buffer. The second type contains format specifiers that define the way the subsequent arguments, specified by *args*, are displayed. For complete information on formatting output, including a description of the format specifiers, see the **Formatter** class in Chapter 20.

Because **System.out** is a **PrintStream**, you can call **printf( )** on **System.out**. Thus, **printf( )** can be used in place of **println( )** when writing to the console whenever formatted output is desired. For example, the following program uses **printf( )** to output numeric values in various formats. In the past, such formatting required a bit of work. With the addition of **printf( )**, this is now an easy task.

```
// Demonstrate printf().

class PrintfDemo {
  public static void main(String args[]) {
    System.out.println("Here are some numeric values " +
                       "in different formats.\n");

    System.out.printf("Various integer formats: ");
    System.out.printf("%d %(d %+d %05d\n", 3, -3, 3, 3);

    System.out.println();
    System.out.printf("Default floating-point format: %f\n",
                      1234567.123);
    System.out.printf("Floating-point with commas: %,f\n",
                      1234567.123);
    System.out.printf("Negative floating-point default: %,f\n",
                      -1234567.123);
    System.out.printf("Negative floating-point option: %,(f\n",
                      -1234567.123);

    System.out.println();

    System.out.printf("Line up positive and negative values:\n");
    System.out.printf("% ,.2f\n% ,.2f\n",
                      1234567.123, -1234567.123);
  }
}
```

The output is shown here:

```
Here are some numeric values in different formats.

Various integer formats: 3 (3) +3 00003

Default floating-point format: 1234567.123000
Floating-point with commas: 1,234,567.123000
Negative floating-point default: -1,234,567.123000
Negative floating-point option: (1,234,567.123000)

Line up positive and negative values:
1,234,567.12
-1,234,567.12
```

PrintStream also defines the **format( )** method. It has these general forms:

PrintStream format(String *fmtString*, Object … *args*)
PrintStream format(Locale *loc*, String *fmtString*, Object … *args*)

It works exactly like **printf( )**.

DataOutputStream and DataInputStream

DataOutputStream and **DataInputStream** enable you to write or read primitive data to or from a stream. They implement the **DataOutput** and **DataInput** interfaces, respectively. These interfaces define methods that convert primitive values to or from a sequence of bytes. These streams make it easy to store binary data, such as integers or floating-point values, in a file. Each is examined here.

DataOutputStream extends **FilterOutputStream**, which extends **OutputStream**. In addition to implementing **DataOutput**, **DataOutputStream** also implements **AutoCloseable**, **Closeable**, and **Flushable**. **DataOutputStream** defines the following constructor:

DataOutputStream(OutputStream *outputStream*)

Here, *outputStream* specifies the output stream to which data will be written. When a **DataOutputStream** is closed (by calling **close( )**), the underlying stream specified by *outputStream* is also closed automatically.

DataOutputStream supports all of the methods defined by its superclasses. However, it is the methods defined by the **DataOutput** interface, which it implements, that make it interesting. **DataOutput** defines methods that convert values of a primitive type into a byte sequence and then writes it to the underlying stream. Here is a sampling of these methods:

final void writeDouble(double *value*) throws IOException
final void writeBoolean(boolean *value*) throws IOException
final void writeInt(int *value*) throws IOException

Here, *value* is the value written to the stream.

DataInputStream is the complement of **DataOuputStream**. It extends **FilterInputStream**, which extends **InputStream**. In addition to implementing the **DataInput** interface, **DataInputStream** also implements **AutoCloseable** and **Closeable**. Here is its only constructor:

DataInputStream(InputStream *inputStream*)

Here, *inputStream* specifies the input stream from which data will be read. When a **DataInputStream** is closed (by calling **close( )**), the underlying stream specified by *inputStream* is also closed automatically.

Like **DataOutputStream**, **DataInputStream** supports all of the methods of its superclasses, but it is the methods defined by the **DataInput** interface that make it unique. These methods read a sequence of bytes and convert them into values of a primitive type. Here is a sampling of these methods:

```
final double readDouble( ) throws IOException
final boolean readBoolean( ) throws IOException
final int readInt( ) throws IOException
```

The following program demonstrates the use of **DataOutputStream** and **DataInputStream**:

```
// Demonstrate DataInputStream and DataOutputStream.

import java.io.*;

class DataIODemo {
  public static void main(String args[]) throws IOException {

    // First, write the data.
    try ( DataOutputStream dout =
            new DataOutputStream(new FileOutputStream("Test.dat")) )
    {
      dout.writeDouble(98.6);
      dout.writeInt(1000);
      dout.writeBoolean(true);

    } catch(FileNotFoundException e) {
      System.out.println("Cannot Open Output File");
      return;
    } catch(IOException e) {
      System.out.println("I/O Error: " + e);
    }

    // Now, read the data back.
    try ( DataInputStream din =
            new DataInputStream(new FileInputStream("Test.dat")) )
    {

      double d = din.readDouble();
      int i = din.readInt();
      boolean b = din.readBoolean();

      System.out.println("Here are the values: " +
                         d + " " + i + " " + b);
    } catch(FileNotFoundException e) {
      System.out.println("Cannot Open Input File");
      return;
    } catch(IOException e) {
      System.out.println("I/O Error: " + e);
    }
  }
}
```

The output is shown here:

```
Here are the values: 98.6 1000 true
```

RandomAccessFile

RandomAccessFile encapsulates a random-access file. It is not derived from **InputStream** or **OutputStream**. Instead, it implements the interfaces **DataInput** and **DataOutput**, which define the basic I/O methods. It also implements the **AutoCloseable** and **Closeable** interfaces. **RandomAccessFile** is special because it supports positioning requests—that is, you can position the file pointer within the file. It has these two constructors:

RandomAccessFile(File *fileObj*, String *access*)
    throws FileNotFoundException
RandomAccessFile(String *filename*, String *access*)
    throws FileNotFoundException

In the first form, *fileObj* specifies the file to open as a **File** object. In the second form, the name of the file is passed in *filename*. In both cases, *access* determines what type of file access is permitted. If it is "r", then the file can be read, but not written. If it is "rw", then the file is opened in read-write mode. If it is "rws", the file is opened for read-write operations and every change to the file's data or metadata will be immediately written to the physical device. If it is "rwd", the file is opened for read-write operations and every change to the file's data will be immediately written to the physical device.

The method **seek( )**, shown here, is used to set the current position of the file pointer within the file:

void seek(long *newPos*) throws IOException

Here, *newPos* specifies the new position, in bytes, of the file pointer from the beginning of the file. After a call to **seek( )**, the next read or write operation will occur at the new file position.

RandomAccessFile implements the standard input and output methods, which you can use to read and write to random access files. It also includes some additional methods. One is **setLength( )**. It has this signature:

void setLength(long *len*) throws IOException

This method sets the length of the invoking file to that specified by *len*. This method can be used to lengthen or shorten a file. If the file is lengthened, the added portion is undefined.

The Character Streams

While the byte stream classes provide sufficient functionality to handle any type of I/O operation, they cannot work directly with Unicode characters. Since one of the main purposes of Java is to support the "write once, run anywhere" philosophy, it was necessary to include direct I/O support for characters. In this section, several of the character I/O classes are discussed. As explained earlier, at the top of the character stream hierarchies are the **Reader** and **Writer** abstract classes. We will begin with them.

Reader

Reader is an abstract class that defines Java’s model of streaming character input. It implements the **AutoCloseable**, **Closeable**, and **Readable** interfaces. All of the methods in this class (except for **markSupported( )**) will throw an **IOException** on error conditions. Table 21-3 provides a synopsis of the methods in **Reader**.

Method	Description
abstract void close( )	Closes the input source. Further read attempts will generate an **IOException**.
void mark(int *numChars*)	Places a mark at the current point in the input stream that will remain valid until *numChars* characters are read.
boolean markSupported( )	Returns **true** if **mark( )/reset( )** are supported on this stream.
static Reader nullReader( )	Returns an open, but null reader, which is a reader that contains no data. Thus, the reader is always at the end of the stream and no input can be obtained. The reader can, however, be closed. (Added by JDK 11.)
int read( )	Returns an integer representation of the next available character from the invoking input stream. –1 is returned when an attempt is made to read at the end of the stream.
int read(char *buffer*[ ])	Attempts to read up to *buffer.length* characters into *buffer* and returns the actual number of characters that were successfully read. –1 is returned when an attempt is made to read at the end of the stream.
int read(CharBuffer *buffer*)	Attempts to read characters into *buffer* and returns the actual number of characters that were successfully read. –1 is returned when an attempt is made to read at the end of the stream.
abstract int read(char *buffer*[ ], int *offset*, int *numChars*)	Attempts to read up to *numChars* characters into *buffer* starting at *buffer*[*offset*], returning the number of characters successfully read. –1 is returned when an attempt is made to read at the end of the stream.
boolean ready( )	Returns **true** if the next input request will not wait. Otherwise, it returns **false**.
void reset( )	Resets the input pointer to the previously set mark.
long skip(long *numChars*)	Skips over *numChars* characters of input, returning the number of characters actually skipped.
long transferTo(Writer *writer*)	Copies the contents of the invoking reader to *writer*, returning the number of characters copied.

Table 21-3 The Methods Defined by **Reader**

Writer

Writer is an abstract class that defines streaming character output. It implements the **AutoCloseable**, **Closeable**, **Flushable**, and **Appendable** interfaces. All of the methods in this class throw an **IOException** in the case of errors. Table 21-4 shows a synopsis of the methods in **Writer**.

Method	Description
Writer append(char *ch*)	Appends *ch* to the end of the invoking output stream. Returns a reference to the invoking stream.
Writer append(CharSequence *chars*)	Appends *chars* to the end of the invoking output stream. Returns a reference to the invoking stream.
Writer append(CharSequence *chars*, int *begin*, int *end*)	Appends the subrange of *chars* specified by *begin* and *end*–1 to the end of the invoking output stream. Returns a reference to the invoking stream.
abstract void close( )	Closes the output stream. Further write attempts will generate an **IOException**.
abstract void flush( )	Finalizes the output state so that any buffers are cleared. That is, it flushes the output buffers.
static Writer nullWriter( )	Returns an open, but null writer, which is a writer to which no output is actually written. Thus, its output methods can be called but don't actually produce output. The writer can, however, be closed. (Added by JDK 11.)
void write(int *ch*)	Writes a single character to the invoking output stream. Note that the parameter is an **int**, which allows you to call **write** with an expression without having to cast it back to **char**. However, only the low-order 16 bits are written.
void write(char *buffer*[ ])	Writes a complete array of characters to the invoking output stream.
abstract void write(char *buffer*[ ], int *offset*, int *numChars*)	Writes a subrange of *numChars* characters from the array *buffer*, beginning at *buffer*[*offset*] to the invoking output stream.
void write(String *str*)	Writes *str* to the invoking output stream.
void write(String *str*, int *offset*, int *numChars*)	Writes a subrange of *numChars* characters from the string *str*, beginning at the specified *offset*.

Table 21-4 The Methods Defined by **Writer**

FileReader

The **FileReader** class creates a **Reader** that you can use to read the contents of a file. Two commonly used constructors are shown here:

FileReader(String *filePath*)
FileReader(File *fileObj*)

Either can throw a **FileNotFoundException**. Here, *filePath* is the full path name of a file, and *fileObj* is a **File** object that describes the file.

The following example shows how to read lines from a file and display them on the standard output device. It reads its own source file, which must be in the current directory.

```
// Demonstrate FileReader.

import java.io.*;

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

    try ( FileReader fr = new FileReader("FileReaderDemo.java") )
    {
      int c;

      // Read and display the file.
      while((c = fr.read()) != -1) System.out.print((char) c);

    } catch(IOException e) {
      System.out.println("I/O Error: " + e);
    }
  }
}
```

FileWriter

FileWriter creates a **Writer** that you can use to write to a file. Four commonly used constructors are shown here:

FileWriter(String *filePath*)
FileWriter(String *filePath*, boolean *append*)
FileWriter(File *fileObj*)
FileWriter(File *fileObj*, boolean *append*)

They can all throw an **IOException**. Here, *filePath* is the full path name of a file, and *fileObj* is a **File** object that describes the file. *If append* is **true**, then output is appended to the end of the file.

Creation of a **FileWriter** is not dependent on the file already existing. **FileWriter** will create the file before opening it for output when you create the object. In the case where you attempt to open a read-only file, an **IOException** will be thrown.

The following example is a character stream version of an example shown earlier when **FileOutputStream** was discussed. This version creates a sample buffer of characters by first making a **String** and then using the **getChars( )** method to extract the character array equivalent. It then creates three files. The first, **file1.txt**, will contain every other character from the sample. The second, **file2.txt**, will contain the entire set of characters. Finally, the third, **file3.txt**, will contain only the last quarter.

```
// Demonstrate FileWriter.

import java.io.*;

class FileWriterDemo {
  public static void main(String args[]) throws IOException {
    String source = "Now is the time for all good men\n"
      + " to come to the aid of their country\n"
      + " and pay their due taxes.";
    char buffer[] = new char[source.length()];
    source.getChars(0, source.length(), buffer, 0);

    try ( FileWriter f0 = new FileWriter("file1.txt");
          FileWriter f1 = new FileWriter("file2.txt");
          FileWriter f2 = new FileWriter("file3.txt") )
    {
      // write to first file
      for (int i=0; i < buffer.length; i += 2) {
        f0.write(buffer[i]);
      }

      // write to second file
      f1.write(buffer);

      // write to third file
      f2.write(buffer,buffer.length-buffer.length/4,buffer.length/4);

    } catch(IOException e) {
      System.out.println("An I/O Error Occurred");
    }
  }
}
```

CharArrayReader

CharArrayReader is an implementation of an input stream that uses a character array as the source. This class has two constructors, each of which requires a character array to provide the data source:

CharArrayReader(char *array* [ ])
CharArrayReader(char *array* [ ], int *start*, int *numChars*)

Here, *array* is the input source. The second constructor creates a **Reader** from a subset of your character array that begins with the character at the index specified by *start* and is *numChars* long.

The **close( )** method implemented by **CharArrayReader** does not throw any exceptions. This is because it cannot fail.

The following example uses a pair of **CharArrayReader**s:

```
// Demonstrate CharArrayReader.

import java.io.*;

public class CharArrayReaderDemo {
  public static void main(String args[]) {
    String tmp = "abcdefghijklmnopqrstuvwxyz";
    int length = tmp.length();
    char c[] = new char[length];
    tmp.getChars(0, length, c, 0);
    int i;

    try (CharArrayReader input1 = new CharArrayReader(c) )
    {
      System.out.println("input1 is:");
      while((i = input1.read()) != -1) {
        System.out.print((char)i);
      }
      System.out.println();
    } catch(IOException e) {
      System.out.println("I/O Error: " + e);
    }

    try ( CharArrayReader input2 = new CharArrayReader(c, 0, 5) )
    {
      System.out.println("input2 is:");
      while((i = input2.read()) != -1) {
        System.out.print((char)i);
      }
      System.out.println();
    } catch(IOException e) {
      System.out.println("I/O Error: " + e);
    }
  }
}
```

The **input1** object is constructed using the entire lowercase alphabet, whereas **input2** contains only the first five letters. Here is the output:

```
input1 is:
abcdefghijklmnopqrstuvwxyz
input2 is:
abcde
```

CharArrayWriter

CharArrayWriter is an implementation of an output stream that uses an array as the destination. **CharArrayWriter** has two constructors, shown here:

CharArrayWriter( )
CharArrayWriter(int *numChars*)

In the first form, a buffer with a default size is created. In the second, a buffer is created with a size equal to that specified by *numChars*. The buffer is held in the **buf** field of **CharArrayWriter**. The buffer size will be increased automatically, if needed. The number of characters held by the buffer is contained in the **count** field of **CharArrayWriter**. Both **buf** and **count** are protected fields.

The **close( )** method has no effect on a **CharArrayWriter**.

The following example demonstrates **CharArrayWriter** by reworking the sample program shown earlier for **ByteArrayOutputStream**. It produces the same output as the previous version.

```
// Demonstrate CharArrayWriter.

import java.io.*;

class CharArrayWriterDemo {
  public static void main(String args[]) throws IOException {
    CharArrayWriter f = new CharArrayWriter();
    String s = "This should end up in the array";
    char buf[] = new char[s.length()];

    s.getChars(0, s.length(), buf, 0);

    try {
      f.write(buf);
    } catch(IOException e) {
      System.out.println("Error Writing to Buffer");
      return;
    }

    System.out.println("Buffer as a string");
    System.out.println(f.toString());
    System.out.println("Into array");

    char c[] = f.toCharArray();
    for (int i=0; i<c.length; i++) {
      System.out.print(c[i]);
    }

    System.out.println("\nTo a FileWriter()");

    // Use try-with-resources to manage the file stream.
    try ( FileWriter f2 = new FileWriter("test.txt") )
    {
      f.writeTo(f2);
    } catch(IOException e) {
      System.out.println("I/O Error: " + e);
    }

    System.out.println("Doing a reset");
    f.reset();

    for (int i=0; i<3; i++) f.write('X');

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

BufferedReader

BufferedReader improves performance by buffering input. It has two constructors:

BufferedReader(Reader *inputStream*)
BufferedReader(Reader *inputStream*, int *bufSize*)

The first form creates a buffered character stream using a default buffer size. In the second, the size of the buffer is passed in *bufSize*.

Closing a **BufferedReader** also causes the underlying stream specified by *inputStream* to be closed.

As is the case with the byte-oriented stream, buffering an input character stream also provides the foundation required to support moving backward in the stream within the available buffer. To support this, **BufferedReader** implements the **mark( )** and **reset( )** methods, and **BufferedReader.markSupported( )** returns **true**. A relatively recent addition to **BufferedReader** is called **lines( )**. It returns a **Stream** reference to the sequence of lines read by the reader. (**Stream** is part of the stream API discussed in Chapter 29.)

The following example reworks the **BufferedInputStream** example, shown earlier, so that it uses a **BufferedReader** character stream rather than a buffered byte stream. As before, it uses the **mark( )** and **reset( )** methods to parse a stream for the HTML entity reference for the copyright symbol. Such a reference begins with an ampersand (&) and ends with a semicolon (;) without any intervening whitespace. The sample input has two ampersands to show the case where the **reset( )** happens and where it does not. Output is the same as that shown earlier.

```
// Use buffered input.

import java.io.*;

class BufferedReaderDemo {
  public static void main(String args[]) throws IOException {
    String s = "This is a &copy; copyright symbol " +
      "but this is &copy not.\n";
    char buf[] = new char[s.length()];
    s.getChars(0, s.length(), buf, 0);

    CharArrayReader in = new CharArrayReader(buf);
    int c;
    boolean marked = false;

    try ( BufferedReader f = new BufferedReader(in) )
    {

      while ((c = f.read()) != -1) {
        switch(c) {
        case '&':
          if (!marked) {
            f.mark(32);
            marked = true;
          } else {
            marked = false;
          }
          break;
        case ';':
          if (marked) {
            marked = false;
            System.out.print("(c)");
          } else

            System.out.print((char) c);
          break;
        case ' ':
          if (marked) {
            marked = false;
            f.reset();
            System.out.print("&");
          } else
            System.out.print((char) c);
          break;
        default:
          if (!marked)
            System.out.print((char) c);
          break;
        }
      }
    } catch(IOException e) {
      System.out.println("I/O Error: " + e);
    }
  }
}
```

BufferedWriter

A **BufferedWriter** is a **Writer** that buffers output. Using a **BufferedWriter** can improve performance by reducing the number of times data is actually physically written to the output device.

A **BufferedWriter** has these two constructors:

BufferedWriter(Writer *outputStream*)
BufferedWriter(Writer *outputStream*, int *bufSize*)

The first form creates a buffered stream using a buffer with a default size. In the second, the size of the buffer is passed in *bufSize*.

PushbackReader

The **PushbackReader** class allows one or more characters to be returned to the input stream. This allows you to look ahead in the input stream. Here are its two constructors:

PushbackReader(Reader *inputStream*)
PushbackReader(Reader *inputStream*, int *bufSize*)

The first form creates a buffered stream that allows one character to be pushed back. In the second, the size of the pushback buffer is passed in *bufSize*.

Closing a **PushbackReader** also closes the underlying stream specified by *inputStream*.

PushbackReader provides **unread( )**, which returns one or more characters to the invoking input stream. It has the three forms shown here:

void unread(int *ch*) throws IOException
void unread(char *buffer* [ ]) throws IOException
void unread(char *buffer* [ ], int *offset*, int *numChars*) throws IOException

The first form pushes back the character passed in *ch*. This will be the next character returned by a subsequent call to **read( )**. The second form returns the characters in *buffer*. The third form pushes back *numChars* characters beginning at *offset* from *buffer*. An **IOException** will be thrown if there is an attempt to return a character when the pushback buffer is full.

The following program reworks the earlier **PushbackInputStream** example by replacing **PushbackInputStream** with **PushbackReader**. As before, it

shows how a programming language parser can use a pushback stream to deal with the difference between the == operator for comparison and the = operator for assignment.

```
// Demonstrate unread().

import java.io.*;

class PushbackReaderDemo {
  public static void main(String args[]) {
    String s = "if (a == 4) a = 0;\n";
    char buf[] = new char[s.length()];
    s.getChars(0, s.length(), buf, 0);
    CharArrayReader in = new CharArrayReader(buf);

    int c;

    try ( PushbackReader f = new PushbackReader(in) )
    {
      while ((c = f.read()) != -1) {
        switch(c) {
        case '=':
          if ((c = f.read()) == '=')
            System.out.print(".eq.");
          else {
            System.out.print("<-");
            f.unread(c);
          }
          break;
        default:
          System.out.print((char) c);
          break;
        }
      }
    } catch(IOException e) {
      System.out.println("I/O Error: " + e);
    }
  }
}
```

PrintWriter

PrintWriter is essentially a character-oriented version of **PrintStream**. It implements the **Appendable**, **AutoCloseable**, **Closeable**, and **Flushable** interfaces. **PrintWriter** has several constructors. The following have been supplied by **PrintWriter** from the start:

PrintWriter(OutputStream *outputStream*)
PrintWriter(OutputStream *outputStream*, boolean *autoFlushingOn*)
PrintWriter(Writer *outputStream*)
PrintWriter(Writer *outputStream*, boolean *autoFlushingOn*)

Here, *outputStream* specifies an open **OutputStream** that will receive output. The *autoFlushingOn* parameter controls whether the output buffer is automatically flushed every time **println( )**, **printf( )**, or **format( )** is called. If *autoFlushingOn* is **true**, flushing automatically takes place. If **false**, flushing is not automatic. Constructors that do not specify the *autoFlushingOn* parameter do not automatically flush.

The next set of constructors gives you an easy way to construct a **PrintWriter** that writes its output to a file.

PrintWriter(File *outputFile*) throws FileNotFoundException
PrintWriter(File *outputFile*, String *charSet*)
throws FileNotFoundException, UnsupportedEncodingException
PrintWriter(String *outputFileName*) throws FileNotFoundException
PrintWriter(String *outputFileName*, String *charSet*)
throws FileNotFoundException, UnsupportedEncodingException

These allow a **PrintWriter** to be created from a **File** object or by specifying the name of a file. In either case, the file is automatically created. Any preexisting file by the same name is destroyed. Once created, the **PrintWriter** object directs all output to the specified file. You can specify a character encoding by passing its name in *charSet*. JDK 11 adds constructors that let you specify a **Charset** parameter.

PrintWriter supports the **print( )** and **println( )** methods for all types, including **Object**. If an argument is not a primitive type, the **PrintWriter** methods will call the object's **toString( )** method and then output the result.

PrintWriter also supports the **printf( )** method. It works the same way it does in the **PrintStream** class described earlier: It allows you to specify the precise format of the data. Here is how **printf( )** is declared in **PrintWriter**:

PrintWriter printf(String *fmtString*, Object … *args*)

PrintWriter printf(Locale *loc*, String *fmtString*, Object …*args*)

The first version writes *args* to standard output in the format specified by *fmtString*, using the default locale. The second lets you specify a locale. Both return the invoking **PrintWriter**.

The **format( )** method is also supported. It has these general forms:

PrintWriter format(String *fmtString*, Object … *args*)
PrintWriter format(Locale *loc*, String *fmtString*, Object … *args*)

It works exactly like **printf( )**.

The Console Class

The **Console** class is used to read from and write to the console, if one exists. It implements the **Flushable** interface. **Console** is primarily a convenience class because most of its functionality is available through **System.in** and **System.out**. However, its use can simplify some types of console interactions, especially when reading strings from the console.

Console supplies no constructors. Instead, a **Console** object is obtained by calling **System.console( )**, which is shown here:

static Console console( )

If a console is available, then a reference to it is returned. Otherwise, **null** is returned. A console will not be available in all cases. Thus, if **null** is returned, no console I/O is possible.

Console defines the methods shown in Table 21-5. Notice that the input methods, such as **readLine( )**, throw **IOError** if an input error occurs. **IOError** is a subclass of **Error**. It indicates an I/O failure that is beyond the control of your program. Thus, you will not normally catch an **IOError**. Frankly, if an **IOError** is thrown while accessing the console, it usually means there has been a catastrophic system failure.

Method	Description
void flush( )	Causes buffered output to be written physically to the console.
Console format(String *fmtString*, Object...*args*)	Writes *args* to the console using the format specified by *fmtString*.
Console printf(String *fmtString*, Object...*args*)	Writes *args* to the console using the format specified by *fmtString*.
Reader reader( )	Returns a reference to a **Reader** connected to the console.
String readLine( )	Reads and returns a string entered at the keyboard. Input stops when the user presses ENTER. If the end of the console input stream has been reached, **null** is returned. An **IOError** is thrown on failure.
String readLine(String *fmtString*, Object...*args*)	Displays a prompting string formatted as specified by *fmtString* and *args*, and then reads and returns a string entered at the keyboard. Input stops when the user presses ENTER. If the end of the console input stream has been reached, **null** is returned. An **IOError** is thrown on failure.
char[ ] readPassword( )	Reads a string entered at the keyboard. Input stops when the user presses ENTER. The string is not displayed. If the end of the console input stream has been reached, **null** is returned. An **IOError** is thrown on failure.
char[ ] readPassword(String *fmtString*, Object... *args*)	Displays a prompting string formatted as specified by *fmtString* and *args*, and then reads a string entered at the keyboard. Input stops when the user presses ENTER. The string is not displayed. If the end of the console input stream has been reached, **null** is returned. An **IOError** is thrown on failure.
PrintWriter writer( )	Returns a reference to a **Writer** connected to the console.

Table 21-5 The Methods Defined by **Console**

Also notice the **readPassword( )** methods. These methods let your application read a password without echoing what is typed. After reading passwords, you should "zero-out" both the array that holds the string entered by the user and the array that holds the password that the string is tested against. This reduces the chance that a malicious program will be able to obtain a password by scanning memory.

Here is an example that demonstrates the **Console** class:

```
// Demonstrate Console.
import java.io.*;

class ConsoleDemo {
  public static void main(String args[]) {
    String str;
    Console con;

    // Obtain a reference to the console.
    con = System.console();
    // If no console available, exit.
    if(con == null) return;

    // Read a string and then display it.
    str = con.readLine("Enter a string: ");
    con.printf("Here is your string: %s\n", str);
  }
}
```

Here is sample output:

```
Enter a string: This is a test.
Here is your string: This is a test.
```

Serialization

Serialization is the process of writing the state of an object to a byte stream. This is useful when you want to save the state of your program to a persistent storage area, such as a file. At a later time, you may restore these objects by using the process of deserialization.

Serialization is also needed to implement *Remote Method Invocation (RMI)*. RMI allows a Java object on one machine to invoke a method of a Java object on a different machine. An object may be supplied as an argument to that remote method. The sending machine serializes the object and transmits it. The receiving machine deserializes it. (More information about RMI appears in Chapter 30.)

Assume that an object to be serialized has references to other objects, which, in turn, have references to still more objects. This set of objects and the relationships among them form a directed graph. There may also be circular references within this object graph. That is, object X may contain a reference to

object Y, and object Y may contain a reference back to object X. Objects may also contain references to themselves. The object serialization and deserialization facilities have been designed to work correctly in these scenarios. If you attempt to serialize an object at the top of an object graph, all of the other referenced objects are recursively located and serialized. Similarly, during the process of deserialization, all of these objects and their references are correctly restored. It is important to note that serialization and deserialization can impact security, especially as it relates to the deserialization of items that you do not trust. Consult the Java documentation for information about this and about security in general.

An overview of the interfaces and classes that support serialization follows.

Serializable

Only an object that implements the **Serializable** interface can be saved and restored by the serialization facilities. The **Serializable** interface defines no members. It is simply used to indicate that a class may be serialized. If a class is serializable, all of its subclasses are also serializable.

Variables that are declared as **transient** are not saved by the serialization facilities. Also, **static** variables are not saved.

Externalizable

The Java facilities for serialization and deserialization have been designed so that much of the work to save and restore the state of an object occurs automatically. However, there are cases in which the programmer may need to have control over these processes. For example, it may be desirable to use compression or encryption techniques. The **Externalizable** interface is designed for these situations.

The **Externalizable** interface defines these two methods:

```
void readExternal(ObjectInput inStream) throws IOException,
    ClassNotFoundException
void writeExternal(ObjectOutput outStream) throws IOException
```

In these methods, *inStream* is the byte stream from which the object is to be read, and *outStream* is the byte stream to which the object is to be written.

ObjectOutput

The **ObjectOutput** interface extends the **DataOutput** and **AutoCloseable** interfaces and supports object serialization. It defines the methods shown in Table 21-6. Note especially the **writeObject( )** method. This is called to serialize an object. All of these methods will throw an **IOException** on error conditions.

Method	Description
void close( )	Closes the invoking stream. Further write attempts will generate an **IOException**.
void flush( )	Finalizes the output state so any buffers are cleared. That is, it flushes the output buffers.
void write(byte *buffer*[ ])	Writes an array of bytes to the invoking stream.
void write(byte *buffer*[ ], int *offset*, int *numBytes*)	Writes a subrange of *numBytes* bytes from the array *buffer*, beginning at *buffer*[*offset*].
void write(int *b*)	Writes a single byte to the invoking stream. The byte written is the low-order byte of *b*.
void writeObject(Object *obj*)	Writes object *obj* to the invoking stream.

Table 21-6 The Methods Defined by **ObjectOutput**

ObjectOutputStream

The **ObjectOutputStream** class extends the **OutputStream** class and implements the **ObjectOutput** interface. It is responsible for writing objects to a stream. One constructor of this class is shown here:

ObjectOutputStream(OutputStream *outStream*) throws IOException

The argument *outStream* is the output stream to which serialized objects will be written. Closing an **ObjectOutputStream** automatically closes the underlying stream specified by *outStream*.

Several commonly used methods in this class are shown in Table 21-7. They will throw an **IOException** on error conditions. There is also an inner class to **ObjectOuputStream** called **PutField**. It facilitates the writing of persistent fields, and its use is beyond the scope of this book.

Method	Description
void close( )	Closes the invoking stream. Further write attempts will generate an **IOException**. The underlying stream is also closed.
void flush( )	Finalizes the output state so any buffers are cleared. That is, it flushes the output buffers.
void write(byte *buffer*[ ])	Writes an array of bytes to the invoking stream.
void write(byte *buffer*[ ], int *offset*, int *numBytes*)	Writes a subrange of *numBytes* bytes from the array *buffer*, beginning at *buffer*[*offset*].
void write(int *b*)	Writes a single **byte** to the invoking stream. The byte written is the low-order byte of *b*.
void writeBoolean(boolean *b*)	Writes a **boolean** to the invoking stream.
void writeByte(int *b*)	Writes a **byte** to the invoking stream. The byte written is the low-order byte of *b*.
void writeBytes(String *str*)	Writes the bytes representing *str* to the invoking stream.
void writeChar(int *c*)	Writes a **char** to the invoking stream.
void writeChars(String *str*)	Writes the characters in *str* to the invoking stream.
void writeDouble(double *d*)	Writes a **double** to the invoking stream.
void writeFloat(float *f*)	Writes a **float** to the invoking stream.
void writeInt(int *i*)	Writes an **int** to the invoking stream.
void writeLong(long *l*)	Writes a **long** to the invoking stream.
final void writeObject(Object *obj*)	Writes *obj* to the invoking stream.
void writeShort(int *i*)	Writes a **short** to the invoking stream.

Table 21-7 A Sampling of Commonly Used Methods Defined by **ObjectOutputStream**

ObjectInput

The **ObjectInput** interface extends the **DataInput** and **AutoCloseable** interfaces and defines the methods shown in Table 21-8. It supports object serialization. Note especially the **readObject( )** method. This is called to deserialize an object. All of these methods will throw an **IOException** on error conditions. The **readObject( )** method can also throw **ClassNotFoundException**.

Method	Description
int available( )	Returns the number of bytes that are now available in the input buffer.
void close( )	Closes the invoking stream. Further read attempts will generate an **IOException**.
int read( )	Returns an integer representation of the next available byte of input. –1 is returned when an attempt is made to read at the end of the stream.
int read(byte *buffer*[ ])	Attempts to read up to *buffer.length* bytes into *buffer*, returning the number of bytes that were successfully read. –1 is returned when an attempt is made to read at the end of the stream.
int read(byte *buffer*[ ], int *offset*, int *numBytes*)	Attempts to read up to *numBytes* bytes into *buffer* starting at *buffer*[*offset*], returning the number of bytes that were successfully read. –1 is returned when an attempt is made to read at the end of the stream.
Object readObject( )	Reads an object from the invoking stream.
long skip(long *numBytes*)	Ignores (that is, skips) *numBytes* bytes in the invoking stream, returning the number of bytes actually ignored.

Table 21-8 The Methods Defined by **ObjectInput**

ObjectInputStream

The **ObjectInputStream** class extends the **InputStream** class and implements the **ObjectInput** interface. **ObjectInputStream** is responsible for reading objects from a stream. One constructor of this class is shown here:

ObjectInputStream(InputStream *inStream*) throws IOException

The argument *inStream* is the input stream from which serialized objects should be read. Closing an **ObjectInputStream** automatically closes the underlying stream specified by *inStream*.

Several commonly used methods in this class are shown in Table 21-9. They will throw an **IOException** on error conditions. The **readObject( )** method can also throw **ClassNotFoundException**. There is also an inner class to **ObjectInputStream** called **GetField**. It facilitates the reading of persistent fields, and its use is beyond the scope of this book.

Method	Description
int available( )	Returns the number of bytes that are now available in the input buffer.
void close( )	Closes the invoking stream. Further read attempts will generate an **IOException**. The underlying stream is also closed.
int read( )	Returns an integer representation of the next available byte of input. –1 is returned when an attempt is made to read at the end of the stream.
int read(byte *buffer*[ ], int *offset*, int *numBytes*)	Attempts to read up to *numBytes* bytes into *buffer* starting at *buffer*[*offset*], returning the number of bytes successfully read. –1 is returned when an attempt is made to read at the end of the stream.
Boolean readBoolean( )	Reads and returns a **boolean** from the invoking stream.
byte readByte( )	Reads and returns a **byte** from the invoking stream.
char readChar( )	Reads and returns a **char** from the invoking stream.
double readDouble( )	Reads and returns a **double** from the invoking stream.
float readFloat( )	Reads and returns a **float** from the invoking stream.
void readFully(byte *buffer*[ ])	Reads *buffer.length* bytes into *buffer*. Returns only when all bytes have been read.
void readFully(byte *buffer*[ ], int *offset*, int *numBytes*)	Reads *numBytes* bytes into *buffer* starting at *buffer*[*offset*]. Returns only when *numBytes* have been read.
int readInt( )	Reads and returns an **int** from the invoking stream.
long readLong( )	Reads and returns a **long** from the invoking stream.
final Object readObject( )	Reads and returns an object from the invoking stream.
short readShort( )	Reads and returns a **short** from the invoking stream.
int readUnsignedByte( )	Reads and returns an unsigned **byte** from the invoking stream.
int readUnsignedShort( )	Reads and returns an unsigned **short** from the invoking stream.

Table 21-9 Commonly Used Methods Defined by **ObjectInputStream**

Beginning with JDK 9, **ObjectInputStream** includes the methods **getObjectInputFilter( )** and **setObjectInputFilter( )**. These support the filtering of object input streams through the use of **ObjectInputFilter**, **ObjectInputFilter.FilterInfo**, **ObjectInputFilter.Config**, and **ObjectInputFilter.Status**, which were all added by JDK 9. Filtering gives you control over deserialization.

A Serialization Example

The following program illustrates how to use object serialization and deserialization. It begins by instantiating an object of class **MyClass**. This object has three instance variables that are of types **String**, **int**, and **double**. This is the information we want to save and restore.

A **FileOutputStream** is created that refers to a file named "serial", and an **ObjectOutputStream** is created for that file stream. The **writeObject( )** method of **ObjectOutputStream** is then used to serialize our object. The object output stream is flushed and closed.

A **FileInputStream** is then created that refers to the file named "serial", and an **ObjectInputStream** is created for that file stream. The **readObject( )** method of **ObjectInputStream** is then used to deserialize our object. The object input stream is then closed.

Note that **MyClass** is defined to implement the **Serializable** interface. If this is not done, a **NotSerializableException** is thrown. Try experimenting with this program by declaring some of the **MyClass** instance variables to be **transient**. That data is then not saved during serialization.

```
// A serialization demo.

import java.io.*;

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

    // Object serialization

    try ( ObjectOutputStream objOStrm =
            new ObjectOutputStream(new FileOutputStream("serial")) )
    {
      MyClass object1 = new MyClass("Hello", -7, 2.7e10);
      System.out.println("object1: " + object1);

      objOStrm.writeObject(object1);
    }
    catch(IOException e) {
      System.out.println("Exception during serialization: " + e);
    }

    // Object deserialization

    try ( ObjectInputStream objIStrm =
            new ObjectInputStream(new FileInputStream("serial")) )
    {
      MyClass object2 = (MyClass)objIStrm.readObject();
      System.out.println("object2: " + object2);
    }
    catch(Exception e) {
      System.out.println("Exception during deserialization: " + e);
    }
  }
}
```

```
class MyClass implements Serializable {
  String s;
  int i;
  double d;

  public MyClass(String s, int i, double d) {
    this.s = s;
    this.i = i;
    this.d = d;
  }

  public String toString() {
    return "s=" + s + "; i=" + i + "; d=" + d;
  }
}
```

This program demonstrates that the instance variables of **object1** and **object2** are identical. The output is shown here:

```
object1: s=Hello; i=-7; d=2.7E10
object2: s=Hello; i=-7; d=2.7E10
```

One last point: For classes that you intend to serialize, you will normally want them to define the **static**, **final**, **long** constant **serialVersionUID** as a private member. Although Java will automatically define this value (as is the case for **MyClass** in the preceding example), for real world applications, it is far better for you to define this value explicitly.

Stream Benefits

The streaming interface to I/O in Java provides a clean abstraction for a complex and often cumbersome task. The composition of the filtered stream classes allows you to dynamically build the custom streaming interface to suit your data transfer requirements. Java programs written to adhere to the abstract, high-level **InputStream**, **OutputStream**, **Reader**, and **Writer** classes will function properly in the future even when new and improved concrete stream classes are invented. As you will see in Chapter 23, this model works very well when we switch from a file system-based set of streams to the network and socket streams. Finally, serialization of objects plays an important role in many types of Java programs. Java's serialization I/O classes provide a portable solution to this

sometimes tricky task.

CHAPTER

22

Exploring NIO

Beginning with version 1.4, Java has provided a second I/O system called NIO (which is short for *New I/O*). It supports a buffer-oriented, channel-based approach to I/O operations. With the release of JDK 7, the NIO system was greatly expanded, providing enhanced support for file-handling and file system features. In fact, so significant were the changes that the term *NIO.2* is often used. Because of the capabilities supported by the NIO file classes, NIO is expected to become an increasingly important approach to file handling. This chapter explores several of the key features of the NIO system.

The NIO Classes

The NIO classes are contained in the packages shown here. Beginning with JDK 9, all are in the **java.base** module.

Package	Purpose
java.nio	Top-level package for the NIO system. Encapsulates various types of buffers that contain data operated upon by the NIO system.
java.nio.channels	Supports channels, which are essentially open I/O connections.
java.nio.channels.spi	Supports service providers for channels.
java.nio.charset	Encapsulates character sets. Also supports encoders and decoders that convert characters to bytes and bytes to characters, respectively.
java.nio.charset.spi	Supports service providers for character sets.
java.nio.file	Provides support for files.
java.nio.file.attribute	Provides support for file attributes.
java.nio.file.spi	Supports service providers for file systems.

Before we begin, it is important to emphasize that the NIO subsystem does not replace the stream-based I/O classes found in **java.io**, which are discussed in Chapter 21, and good working knowledge of the stream-based I/O in **java.io** is helpful to understanding NIO.

NOTE This chapter assumes that you have read the overview of I/O given in Chapter 13 and the discussion of stream-based I/O supplied in Chapter 21.

NIO Fundamentals

The NIO system is built on two foundational items: buffers and channels. A *buffer* holds data. A *channel* represents an open connection to an I/O device, such as a file or a socket. In general, to use the NIO system, you obtain a channel to an I/O device and a buffer to hold data. You then operate on the buffer, inputting or outputting data as needed. The following sections examine buffers and channels in more detail.

Buffers

Buffers are defined in the **java.nio** package. All buffers are subclasses of the **Buffer** class, which defines the core functionality common to all buffers: current position, limit, and capacity. The *current position* is the index within the buffer at which the next read or write operation will take place. The current position is advanced by most read or write operations. The *limit* is the index value one past the last valid location in the buffer. The *capacity* is the number of elements that the buffer can hold. Often the limit equals the capacity of the buffer. **Buffer** also supports mark and reset. **Buffer** defines several methods, which are shown in Table 22-1.

Method	Description
abstract Object array( )	If the invoking buffer is backed by an array, returns a reference to the array. Otherwise, an **UnsupportedOperationException** is thrown. If the array is read-only, a **ReadOnlyBufferException** is thrown.
abstract int arrayOffset( )	If the invoking buffer is backed by an array, returns the index of the first element. Otherwise, an **UnsupportedOperationException** is thrown. If the array is read-only, a **ReadOnlyBufferException** is thrown.
final int capacity( )	Returns the number of elements that the invoking buffer is capable of holding.
final Buffer clear( )	Clears the invoking buffer and returns a reference to the buffer.
abstract Buffer duplicate( )	Returns a buffer that is identical to the invoking buffer. Thus, both buffers will contain and refer to the same elements.
final Buffer flip( )	Sets the invoking buffer's limit to the current position and resets the current position to 0. Returns a reference to the buffer.
abstract boolean hasArray( )	Returns **true** if the invoking buffer is backed by a read/write array and **false** otherwise.
final boolean hasRemaining( )	Returns **true** if there are elements remaining in the invoking buffer. Returns **false** otherwise.
abstract boolean isDirect( )	Returns **true** if the invoking buffer is direct, which means I/O operations act directly upon it. Returns **false** otherwise.
abstract boolean isReadOnly( )	Returns **true** if the invoking buffer is read-only. Returns **false** otherwise.
final int limit( )	Returns the invoking buffer's limit.
final Buffer limit(int *n*)	Sets the invoking buffer's limit to *n*. Returns a reference to the buffer.
final Buffer mark( )	Sets the mark and returns a reference to the invoking buffer.
final int position( )	Returns the current position.
final Buffer position(int *n*)	Sets the invoking buffer's current position to *n*. Returns a reference to the buffer.
int remaining( )	Returns the number of elements available before the limit is reached. In other words, it returns the limit minus the current position.
final Buffer reset( )	Resets the current position of the invoking buffer to the previously set mark. Returns a reference to the buffer.
final Buffer rewind( )	Sets the position of the invoking buffer to 0. Returns a reference to the buffer.
abstract Buffer slice( )	Returns a buffer that consists of the elements in the invoking buffer, beginning at the invoking buffer's current position. Thus, for the slice, both buffers will contain and refer to the same elements.

Table 22-1 The Methods Defined by **Buffer**

From **Buffer**, the following specific buffer classes are derived, which hold the type of data that their names imply:

ByteBuffer	CharBuffer	DoubleBuffer	FloatBuffer
IntBuffer	LongBuffer	MappedByteBuffer	ShortBuffer

MappedByteBuffer is a subclass of **ByteBuffer** and is used to map a file to a buffer.

All of the aforementioned buffers provide various **get( )** and **put( )** methods, which allow you to get data from a buffer or put data into a buffer. (Of course, if a buffer is read-only, then **put( )** operations are not available.) Table 22-2 shows the **get( )** and **put( )** methods defined by **ByteBuffer**. The other buffer classes have similar methods. All buffer classes also support methods that perform various buffer operations. For example, you can allocate a buffer manually using **allocate( )**. You can wrap an array inside a buffer using **wrap( )**. You can create a subsequence of a buffer using **slice( )**.

Method	Description
abstract byte get( )	Returns the byte at the current position.
ByteBuffer get(byte *vals*[ ])	Copies the invoking buffer into the array referred to by *vals*. Returns a reference to the buffer. If there are not *vals*.**length** elements remaining in the buffer, a **BufferUnderflowException** is thrown.
ByteBuffer get(byte *vals*[ ], int *start*, int *num*)	Copies *num* elements from the invoking buffer into the array referred to by *vals*, beginning at the index specified by *start*. Returns a reference to the buffer. If there are not *num* elements remaining in the buffer, a **BufferUnderflowException** is thrown.
abstract byte get(int *idx*)	Returns the byte at the index specified by *idx* within the invoking buffer.
abstract ByteBuffer put(byte *b*)	Copies *b* into the invoking buffer at the current position. Returns a reference to the buffer. If the buffer is full, a **BufferOverflowException** is thrown.
final ByteBuffer put(byte *vals*[ ])	Copies all elements of *vals* into the invoking buffer, beginning at the current position. Returns a reference to the buffer. If the buffer cannot hold all of the elements, a **BufferOverflowException** is thrown.
ByteBuffer put(byte *vals*[ ], int *start*, int *num*)	Copies *num* elements from *vals*, beginning at *start*, into the invoking buffer. Returns a reference to the buffer. If the buffer cannot hold all of the elements, a **BufferOverflowException** is thrown.
ByteBuffer put(ByteBuffer *bb*)	Copies the elements in *bb* to the invoking buffer, beginning at the current position. If the buffer cannot hold all of the elements, a **BufferOverflowException** is thrown. Returns a reference to the buffer.
abstract ByteBuffer put(int *idx*, byte *b*)	Copies *b* into the invoking buffer at the location specified by *idx*. Returns a reference to the buffer.

Table 22-2 The **get( )** and **put( )** Methods Defined for **ByteBuffer**

Channels

Channels are defined in **java.nio.channels**. A channel represents an open connection to an I/O source or destination. Channels implement the **Channel** interface. It extends **Closeable**, and it extends **AutoCloseable**. By implementing **AutoCloseable**, channels can be managed with a **try**-with-resources statement. When used in a **try**-with-resources block, a channel is closed automatically when it is no longer needed. (See Chapter 13 for a discussion of **try**-with-

resources.)

One way to obtain a channel is by calling **getChannel( )** on an object that supports channels. For example, **getChannel( )** is supported by the following I/O classes:

DatagramSocket	FileInputStream	FileOutputStream
RandomAccessFile	ServerSocket	Socket

The specific type of channel returned depends upon the type of object **getChannel( )** is called on. For example, when called on a **FileInputStream**, **FileOutputStream**, or **RandomAccessFile**, **getChannel( )** returns a channel of type **FileChannel**. When called on a **Socket**, **getChannel( )** returns a **SocketChannel**.

Another way to obtain a channel is to use one of the **static** methods defined by the **Files** class. For example, using **Files**, you can obtain a byte channel by calling **newByteChannel( )**. It returns a **SeekableByteChannel**, which is an interface implemented by **FileChannel**. (The **Files** class is examined in detail later in this chapter.)

Channels such as **FileChannel** and **SocketChannel** support various **read( )** and **write( )** methods that enable you to perform I/O operations through the channel. For example, here are a few of the **read( )** and **write( )** methods defined for **FileChannel**:

Method	Description
abstract int read(ByteBuffer *bb*) throws IOException	Reads bytes from the invoking channel into *bb* until the buffer is full or there is no more input. Returns the number of bytes actually read. Returns –1 when an attempt is made to read at the end of the file.
abstract int read(ByteBuffer *bb*, long *start*) throws IOException	Beginning at the file location specified by *start*, reads bytes from the invoking channel into *bb* until the buffer is full or there is no more input. The current position is unchanged. Returns the number of bytes actually read or –1 if *start* is beyond the end of the file.
abstract int write(ByteBuffer *bb*) throws IOException	Writes the contents of *bb* to the invoking channel, starting at the current position. Returns the number of bytes written.
abstract int write(ByteBuffer *bb*, long *start*) throws IOException	Beginning at the file location specified by *start*, writes the contents of *bb* to the invoking channel. The current position is unchanged. Returns the number of bytes written.

All channels support additional methods that give you access to and control over the channel. For example, **FileChannel** supports methods to get or set the current position, transfer information between file channels, obtain the current size of the channel, and lock the channel, among others. **FileChannel** provides a **static** method called **open( )**, which opens a file and returns a channel to it. This provides another way to obtain a channel. **FileChannel** also provides the **map( )** method, which lets you map a file to a buffer.

Charsets and Selectors

Two other entities used by NIO are charsets and selectors. A *charset* defines the way that bytes are mapped to characters. You can encode a sequence of characters into bytes using an *encoder*. You can decode a sequence of bytes into characters using a *decoder*. Charsets, encoders, and decoders are supported by classes defined in the **java.nio.charset** package. Because default encoders and decoders are provided, you will not often need to work explicitly with charsets.

A *selector* supports key-based, non-blocking, multiplexed I/O. In other words, selectors enable you to perform I/O through multiple channels. Selectors are supported by classes defined in the **java.nio.channels** package. Selectors are most applicable to socket-backed channels.

We will not use charsets or selectors in this chapter, but you might find them

useful in your own applications.

Enhancements Added by NIO.2

Beginning with JDK 7, the NIO system was substantially expanded and enhanced. In addition to support for the **try**-with-resources statement (which provides automatic resource management), the improvements included three new packages (**java.nio.file**, **java.nio.file.attribute**, and **java.nio.file.spi**); several new classes, interfaces, and methods; and direct support for stream-based I/O. The additions have greatly expanded the ways in which NIO can be used, especially with files. Several of the key additions are described in the following sections.

The Path Interface

Perhaps the single most important addition to the NIO system was the **Path** interface because it encapsulates a path to a file. As you will see, **Path** is the glue that binds together many of the NIO.2 file-based features. It describes a file's location within the directory structure. **Path** is packaged in **java.nio.file**, and it inherits the following interfaces: **Watchable**, **Iterable<Path>**, and **Comparable<Path>**. **Watchable** describes an object that can be monitored for changes. The **Iterable** and **Comparable** interfaces were described earlier in this book.

Path declares a number of methods that operate on the path. A sampling is shown in Table 22-3. Pay special attention to the **getName( )** method. It is used to obtain an element in a path. It works using an index. At index zero is the part of the path nearest the root, which is the leftmost element in a path. Subsequent indexes specify elements to the right of the root. The number of elements in a path can be obtained by calling **getNameCount( )**. If you want to obtain a string representation of the entire path, simply call **toString( )**. Notice that you can resolve a relative path into an absolute path by using the **resolve( )** method.

Method	Description
default boolean endsWith(String *path*)	Returns **true** if the invoking **Path** ends with the path specified by *path*. Otherwise, returns **false**.
boolean endsWith(Path *path*)	Returns **true** if the invoking **Path** ends with the path specified by *path*. Otherwise, returns **false**.
Path getFileName( )	Returns the filename associated with the invoking **Path**.
Path getName(int *idx*)	Returns a **Path** object that contains the name of the path element specified by *idx* within the invoking object. The leftmost element is at index 0. This is the element nearest the root. The rightmost element is at **getNameCount( )** – 1.
int getNameCount( )	Returns the number of elements beyond the root directory in the invoking **Path**.
Path getParent( )	Returns a **Path** that contains the entire path except for the name of the file specified by the invoking **Path**.
Path getRoot( )	Returns the root of the invoking **Path**.
boolean isAbsolute( )	Returns **true** if the invoking **Path** is absolute. Otherwise, returns **false**.
static Path of(String *pathname*, String ... *parts*)	Returns a **Path** that encapsulates the specified path. If the *parts* varargs parameter is not used, then the path must be specified in its entirety by *pathname*. Otherwise, the arguments passed via *parts* are added to *pathname* (usually with an appropriate separator) to form the entire path. In either case, if the path specified is syntactically invalid, an **InvalidPathException** will occur. (Added by JDK 11.)
static Path of(URI uri)	The path corresponding to *uri* is returned. (Added by JDK 11.)
Path resolve(Path *path*)	If *path* is absolute, *path* is returned. Otherwise, if *path* does not contain a root, *path* is prefixed by the root specified by the invoking **Path** and the result is returned. If *path* is empty, the invoking **Path** is returned. Otherwise, the behavior is unspecified.
default Path resolve(String *path*)	If *path* is absolute, *path* is returned. Otherwise, if *path* does not contain a root, *path* is prefixed by the root specified by the invoking **Path** and the result is returned. If *path* is empty, the invoking **Path** is returned. Otherwise, the behavior is unspecified.
default boolean startsWith(String *path*)	Returns **true** if the invoking **Path** starts with the path specified by *path*. Otherwise, returns **false**.
boolean startsWith(Path *path*)	Returns **true** if the invoking **Path** starts with the path specified by *path*. Otherwise, returns **false**.
Path toAbsolutePath( )	Returns the invoking **Path** as an absolute path.
String toString( )	Returns a string representation of the invoking **Path**.

Table 22-3 A Sampling of Methods Specified by **Path**

Beginning with JDK 11, an important new **static** factory method called **of( )** was added to **Path**. It returns a **Path** instance from either a path name or a URI. Thus, **of( )** gives you a way to construct a new **Path** instance.

One other point: When updating legacy code that uses the **File** class defined by **java.io**, it is possible to convert a **File** instance into a **Path** instance by calling **toPath( )** on the **File** object. Furthermore, it is possible to obtain a **File** instance by calling the **toFile( )** method defined by **Path**.

The Files Class

Many of the actions that you perform on a file are provided by **static** methods within the **Files** class. The file to be acted upon is specified by its **Path**. Thus, the **Files** methods use a **Path** to specify the file that is being operated upon. **Files** contains a wide array of functionality. For example, it has methods that let you open or create a file that has the specified path. You can obtain information about a **Path**, such as whether it is executable, hidden, or read-only. **Files** also supplies methods that let you copy or move files. A sampling is shown in Table 22-4. In addition to **IOException**, several other exceptions are possible. **Files** also includes these four methods: **list( )**, **walk( )**, **lines( )**, and **find( )**. All return a **Stream** object. These methods help integrate NIO with the stream API described in Chapter 29. Beginning with JDK 11, **Files** also includes the methods **readString( )** and **writeString( )**, which returns a **String** containing the characters in a file or writes a **CharSequence** (such as a **String**) to a file.

Method	Description
static Path copy(Path *src*, Path *dest*, CopyOption ... *how*) throws IOException	Copies the file specified by *src* to the location specified by *dest*. The *how* parameter specifies how the copy will take place.
static Path createDirectory(Path *path*, FileAttribute<?> ... *attribs*) throws IOException	Creates the directory whose path is specified by *path*. The directory attributes are specified by *attribs*.
static Path createFile(Path *path*, FileAttribute<?> ... *attribs*) throws IOException	Creates the file whose path is specified by *path*. The file attributes are specified by *attribs*.
static void delete(Path *path*) throws IOException	Deletes the file whose path is specified by *path*.
static boolean exists(Path *path*, LinkOption ... *opts*)	Returns **true** if the file specified by *path* exists and **false** otherwise. If *opts* is not specified, then symbolic links are followed. To prevent the following of symbolic links, pass **NOFOLLOW_LINKS** to *opts*.
static boolean isDirectory(Path *path*, LinkOption ... *opts*)	Returns **true** if *path* specifies a directory and **false** otherwise. If *opts* is not specified, then symbolic links are followed. To prevent the following of symbolic links, pass **NOFOLLOW_LINKS** to *opts*.
static boolean isExecutable(Path *path*)	Returns **true** if the file specified by *path* is executable and **false** otherwise.
static boolean isHidden(Path *path*) throws IOException	Returns **true** if the file specified by *path* is hidden and **false** otherwise.
static boolean isReadable(Path *path*)	Returns **true** if the file specified by *path* can be read from and **false** otherwise.
static boolean isRegularFile(Path *path*, LinkOption ... *opts*)	Returns **true** if *path* specifies a file and **false** otherwise. If *opts* is not specified, then symbolic links are followed. To prevent the following of symbolic links, pass **NOFOLLOW_LINKS** to *opts*.
static boolean isWritable(Path *path*)	Returns **true** if the file specified by *path* can be written to and **false** otherwise.

static Path move(Path *src*, Path *dest*, CopyOption ... *how*) throws IOException	Moves the file specified by *src* to the location specified by *dest*. The *how* parameter specifies how the move will take place.
static SeekableByteChannel newByteChannel(Path *path*, OpenOption ... *how*) throws IOException	Opens the file specified by *path*, as specified by *how*. Returns a **SeekableByteChannel** to the file. This is a byte channel whose current position can be changed. **SeekableByteChannel** is implemented by **FileChannel**.
static DirectoryStream<Path> newDirectoryStream(Path *path*) throws IOException	Opens the directory specified by *path*. Returns a **DirectoryStream** linked to the directory.
static InputStream newInputStream(Path *path*, OpenOption ... *how*) throws IOException	Opens the file specified by *path*, as specified by *how*. Returns an **InputStream** linked to the file.
static OutputStream newOutputStream(Path *path*, OpenOption ... *how*) throws IOException	Opens the file specified by the invoking object, as specified by *how*. Returns an **OutputStream** linked to the file.
static boolean notExists(Path *path*, LinkOption ... *opts*)	Returns **true** if the file specified by *path* does *not* exist and **false** otherwise. If *opts* is not specified, then symbolic links are followed. To prevent the following of symbolic links, pass **NOFOLLOW_LINKS** to *opts*.
static <A extends BasicFileAttributes> A readAttributes(Path *path*, Class<A> *attribType*, LinkOption ... *opts*) throws IOException	Obtains the attributes associated with the file specified by *path*. The type of attributes to obtain is passed in *attribType*. If *opts* is not specified, then symbolic links are followed. To prevent the following of symbolic links, pass **NOFOLLOW_LINKS** to *opts*.
static long size(Path *path*) throws IOException	Returns the size of the file specified by *path*.

Table 22-4 Commonly Used Listener Interfaces Implemented by Adapter Classes

Notice that several of the methods in Table 22-4 take an argument of type **OpenOption**. This is an interface that describes how to open a file. It is implemented by the **StandardOpenOption** class, which defines an enumeration that has the values shown in Table 22-5.

Value	Meaning
APPEND	Causes output to be written to the end of the file.
CREATE	Creates the file if it does not already exist.
CREATE_NEW	Creates the file only if it does not already exist.
DELETE_ON_CLOSE	Deletes the file when it is closed.
DSYNC	Causes changes to the file to be immediately written to the physical file. Normally, changes to a file are buffered by the file system in the interest of efficiency, being written to the file only as needed.
READ	Opens the file for input operations.
SPARSE	Indicates to the file system that the file is sparse, meaning that it may not be completely filled with data. If the file system does not support sparse files, this option is ignored.
SYNC	Causes changes to the file or its metadata to be immediately written to the physical file. Normally, changes to a file are buffered by the file system in the interest of efficiency, being written to the file only as needed.
TRUNCATE_EXISTING	Causes a preexisting file opened for output to be reduced to zero length.
WRITE	Opens the file for output operations.

Table 22-5 The Standard Open Options

The Paths Class

Because **Path** is an interface, not a class, you can't create an instance of **Path** directly through the use of a constructor. Instead, you obtain a **Path** by a calling a method that returns one. Prior to JDK 11, you would typically do this by using the **get( )** method defined by the **Paths** class. There are two forms of **get( )**. The first is shown here:

static Path get(String *pathname*, String ... *parts*)

It returns a **Path** that encapsulates the specified path. The path can be specified in two ways. First, if *parts* is not used, then the path must be specified in its entirety by *pathname*. Alternatively, you can pass the path in pieces, with the first part passed in *pathname* and the subsequent elements specified by the *parts* varargs parameter. In either case, if the path specified is syntactically invalid, **get( )** will throw an **InvalidPathException**.

The second form of **get( )** creates a **Path** from a **URI**. It is shown here:

static Path get(URI *uri*)

The **Path** corresponding to *uri* is returned.

Although the **Paths.get( )** method just described has been in use since JDK 7 and, at the time of this writing, is still available for use, it is no longer recommended. Instead, the Java API documentation for JDK 11 recommends the use of the new **Path.of( )** method, which was added by JDK 11. Because of this, **Path.of( )** is now the preferred approach. Of course, if you are using a compiler that predates JDK 11, then you must continue to use **Paths.get( )**.

It is important to understand that obtaining a **Path** to a file does not open or create a file. It simply creates an object that encapsulates the file's directory path.

The File Attribute Interfaces

Associated with a file is a set of attributes. These attributes include such things as the file's time of creation, the time of its last modification, whether the file is a directory, and its size. NIO organizes file attributes into several different interfaces. Attributes are represented by a hierarchy of interfaces defined in **java.nio.file.attribute**. At the top is **BasicFileAttributes**. It encapsulates the set of attributes that are commonly found in a variety of file systems. The methods defined by **BasicFileAttributes** are shown in Table 22-6.

Method	Description
FileTime creationTime( )	Returns the time at which the file was created. If creation time is not provided by the file system, then an implementation-dependent value is returned.
Object fileKey( )	Returns the file key. If not supported, **null** is returned.
boolean isDirectory( )	Returns **true** if the file represents a directory.
boolean isOther( )	Returns **true** if the file is not a file, symbolic link, or a directory.
boolean isRegularFile( )	Returns **true** if the file is a normal file, rather than a directory or symbolic link.
boolean isSymbolicLink( )	Returns **true** if the file is a symbolic link.
FileTime lastAccessTime( )	Returns the time at which the file was last accessed. If the time of last access is not provided by the file system, then an implementation-dependent value is returned.
FileTime lastModifiedTime( )	Returns the time at which the file was last modified. If the time of last modification is not provided by the file system, then an implementation-dependent value is returned.
long size( )	Returns the size of the file.

Table 22-6 The Methods Defined by **BasicFileAttributes**

From **BasicFileAttributes** two interfaces are derived: **DosFileAttributes** and **PosixFileAttributes**. **DosFileAttributes** describes those attributes related to the FAT file system as first defined by DOS. It defines the methods shown here:

Method	Description
boolean isArchive( )	Returns **true** if the file is flagged for archiving and **false** otherwise.
boolean isHidden( )	Returns **true** if the file is hidden and **false** otherwise.
boolean isReadOnly( )	Returns **true** if the file is read-only and **false** otherwise.
boolean isSystem( )	Returns **true** if the file is flagged as a system file and **false** otherwise.

PosixFileAttributes encapsulates attributes defined by the POSIX standards. (POSIX stands for *Portable Operating System Interface.*) It defines the methods shown here:

Method	Description
GroupPrincipal group( )	Returns the file's group owner.
UserPrincipal owner( )	Returns the file's owner.
Set<PosixFilePermission> permissions( )	Returns the file's permissions.

There are various ways to access a file's attributes. First, you can obtain an object that encapsulates a file's attributes by calling **readAttributes( )**, which is a **static** method defined by **Files**. One of its forms is shown here:

```
static <A extends BasicFileAttributes>
A readAttributes(Path path, Class<A> attrType, LinkOption... opts)
    throws IOException
```

This method returns a reference to an object that specifies the attributes associated with the file passed in *path*. The specific type of attributes is specified as a **Class** object in the *attrType* parameter. For example, to obtain the basic file attributes, pass **BasicFileAttributes.class** to *attrType*. For DOS attributes, use **DosFileAttributes.class**, and for POSIX attributes, use **PosixFileAttributes.class**. Optional link options are passed via *opts*. If not specified, symbolic links are followed. The method returns a reference to requested attributes. If the requested attribute type is not available, **UnsupportedOperationException** is thrown. Using the object returned, you can access the file's attributes.

A second way to gain access to a file's attributes is to call

getFileAttributeView( ) defined by **Files**. NIO defines several attribute view interfaces, including **AttributeView**, **BasicFileAttributeView**, **DosFileAttributeView**, and **PosixFileAttributeView**, among others. Although we won't be using attribute views in this chapter, they are a feature that you may find helpful in some situations.

In some cases, you won't need to use the file attribute interfaces directly because the **Files** class offers **static** convenience methods that access several of the attributes. For example, **Files** includes methods such as **isHidden( )** and **isWritable( )**.

It is important to understand that not all file systems support all possible attributes. For example, the DOS file attributes apply to the older FAT file system as first defined by DOS. The attributes that will apply to a wide variety of file systems are described by **BasicFileAttributes**. For this reason, these attributes are used in the examples in this chapter.

The FileSystem, FileSystems, and FileStore Classes

You can easily access the file system through the **FileSystem** and **FileSystems** classes packaged in **java.nio.file**. In fact, by using the **newFileSystem( )** method defined by **FileSystems**, it is even possible to obtain a new file system. The **FileStore** class encapsulates the file storage system. Although these classes are not used directly in this chapter, you may find them helpful in your own applications.

Using the NIO System

This section illustrates how to apply the NIO system to a variety of tasks. Before beginning, it is important to emphasize that beginning with JDK 7, the NIO subsystem was greatly expanded. As a result, its uses have also been greatly expanded. As mentioned, the enhanced version is sometimes referred to as NIO.2. Because the features added by NIO.2 are so substantial, they have changed the way that much NIO-based code is written and have increased the types of tasks to which NIO can be applied. Because of its importance, the remaining discussion and examples in this chapter utilize NIO.2 features and, therefore, require a modern version of Java.

In the past, the primary purpose of NIO was channel-based I/O, and this is still a very important use. However, you can now use NIO for stream-based I/O and for performing file-system operations. As a result, the discussion of using

NIO is divided into three parts:

- Using NIO for channel-based I/O
- Using NIO for stream-based I/O
- Using NIO for path and file system operations

Because the most common I/O device is the disk file, the rest of this chapter uses disk files in the examples. Because all file channel operations are byte-based, the type of buffers that we will be using are of type **ByteBuffer**.

Before you can open a file for access via the NIO system, you must obtain a **Path** that describes the file. In the past, one way to do this was to call the **Paths.get( )** factory method, which has been available since JDK 7. However, as explained earlier, beginning with JDK 11, the preferred approach is to use **Path.of( )** rather than **Paths.get( )**. Because of this, the examples use **Path.of( )**. If you are using a version of Java prior to JDK 11, simply substitute **Paths.get( )** for **Path.of( )** in the programs. The form of **of( )** used in the examples is shown here:

static Path of(String *pathname*, String ... *parts*)

Recall that the path can be specified in two ways. It can be passed in pieces, with the first part passed in *pathname* and the subsequent elements specified by the *parts* varargs parameter. Alternatively, the entire path can be specified in *pathname* and *parts* is not used. This is the approach used by the examples.

Use NIO for Channel-Based I/O

An important use of NIO is to access a file via a channel and buffers. The following sections demonstrate some techniques that use a channel to read from and write to a file.

Reading a File via a Channel

There are several ways to read data from a file using a channel. Perhaps the most common way is to manually allocate a buffer and then perform an explicit read operation that loads that buffer with data from the file. It is with this approach that we begin.

Before you can read from a file, you must open it. To do this, first create a **Path** that describes the file. Then use this **Path** to open the file. There are various ways to open the file depending on how it will be used. In this example,

the file will be opened for byte-based input via explicit input operations. Therefore, this example will open the file and establish a channel to it by calling **Files.newByteChannel( )**. The version of **newByteChannel( )** that we will use has this general form:

```
static SeekableByteChannel newByteChannel(Path path, OpenOption ...
    how)
    throws IOException
```

It returns a **SeekableByteChannel** object, which encapsulates the channel for file operations. The **Path** that describes the file is passed in *path*. The *how* parameter specifies how the file will be opened. Because it is a varargs parameter, you can specify zero or more comma-separated arguments. (The valid values were discussed earlier and shown in Table 22-5.) If no arguments are specified, the file is opened for input operations. **SeekableByteChannel** is an interface that describes a channel that can be used for file operations. It is implemented by the **FileChannel** class. When the default file system is used, the returned object can be cast to **FileChannel**. You must close the channel after you have finished with it. Since all channels, including **FileChannel**, implement **AutoCloseable**, you can use a **try**-with-resources statement to close the file automatically instead of calling **close( )** explicitly. This approach is used in the examples.

Next, you must obtain a buffer that will be used by the channel either by wrapping an existing array or by allocating the buffer dynamically. The examples use allocation, but the choice is yours. Because file channels operate on byte buffers, we will use the **allocate( )** method defined by **ByteBuffer** to obtain the buffer. It has this general form:

```
static ByteBuffer allocate(int cap)
```

Here, *cap* specifies the capacity of the buffer. A reference to the buffer is returned.

After you have created the buffer, call **read( )** on the channel, passing a reference to the buffer. The version of **read( )** that we will use is shown next:

```
int read(ByteBuffer buf) throws IOException
```

Each time it is called, **read( )** fills the buffer specified by *buf* with data from the file. The reads are sequential, meaning that each call to **read( )** reads the next buffer's worth of bytes from the file. The **read( )** method returns the number of bytes actually read. It returns –1 when there is an attempt to read at the end of the file.

The following program puts the preceding discussion into action by reading a file called **test.txt** through a channel using explicit input operations:

```
// Use Channel I/O to read a file.

import java.io.*;
import java.nio.*;
import java.nio.channels.*;
import java.nio.file.*;

public class ExplicitChannelRead {
  public static void main(String args[]) {
    int count;
    Path filepath = null;

    // First, obtain a path to the file.
    try {
      filepath = Path.of("test.txt");
    } catch(InvalidPathException e) {
      System.out.println("Path Error " + e);
      return;
    }

    // Next, obtain a channel to that file within a try-with-resources block.
    try ( SeekableByteChannel fChan = Files.newByteChannel(filepath) )
    {

      // Allocate a buffer.
      ByteBuffer mBuf = ByteBuffer.allocate(128);

      do {
        // Read a buffer.
        count = fChan.read(mBuf);

        // Stop when end of file is reached.
        if(count != -1) {

          // Rewind the buffer so that it can be read.
          mBuf.rewind();

          // Read bytes from the buffer and show
          // them on the screen as characters.
          for(int i=0; i < count; i++)
            System.out.print((char)mBuf.get());
        }

      } while(count != -1);

      System.out.println();
    } catch (IOException e) {
      System.out.println("I/O Error " + e);
    }
  }
}
```

Here is how the program works. First, a **Path** object is obtained that contains the relative path to a file called **test.txt**. A reference to this object is assigned to **filepath**. Next, a channel connected to the file is obtained by calling **newByteChannel( )**, passing in **filepath**. Because no open option is specified, the file is opened for reading. Notice that this channel is the object managed by the **try**-with-resources statement. Thus, the channel is automatically closed when the block ends. The program then calls the **allocate( )** method of **ByteBuffer** to allocate a buffer that will hold the contents of the file when it is read. A reference to this buffer is stored in **mBuf**. The contents of the file are then read, one buffer at a time, into **mBuf** through a call to **read( )**. The number of bytes read is stored in **count**. Next, the buffer is rewound through a call to **rewind( )**. This call is necessary because the current position is at the end of the buffer after the call to **read( )**. It must be reset to the start of the buffer in order for the bytes in **mBuf** to be read by calling **get( )**. (Recall that **get( )** is defined by **ByteBuffer**.) Because **mBuf** is a byte buffer, the values returned by **get( )** are bytes. They are cast to **char** so the file can be displayed as text. (Alternatively, it is possible to create a buffer that encodes the bytes into characters and then read that buffer.) When the end of the file has been reached, the value returned by **read( )** will be –1. When this occurs, the program ends, and the channel is automatically closed.

As a point of interest, notice that the program obtains the **Path** within one **try** block and then uses another **try** block to obtain and manage a channel linked to that path. Although there is nothing wrong, per se, with this approach, in many cases, it can be streamlined so that only one **try** block is needed. In this approach, the calls to **Path.of( )** and **newByteChannel( )** are sequenced together. For example, here is a reworked version of the program that uses this approach:

```
// A more compact way to open a channel.

import java.io.*;
import java.nio.*;
import java.nio.channels.*;
import java.nio.file.*;

public class ExplicitChannelRead {
  public static void main(String args[]) {
    int count;

    // Here, the channel is opened on the Path returned by Path.of().
    // There is no need for the filepath variable.
    try ( SeekableByteChannel fChan =
            Files.newByteChannel(Path.of("test.txt")) )
    {
      // Allocate a buffer.
      ByteBuffer mBuf = ByteBuffer.allocate(128);

      do {
        // Read a buffer.
        count = fChan.read(mBuf);

        // Stop when end of file is reached.
        if(count != -1) {

          // Rewind the buffer so that it can be read.
          mBuf.rewind();

          // Read bytes from the buffer and show
          // them on the screen as characters.
          for(int i=0; i < count; i++)
            System.out.print((char)mBuf.get());
        }
      } while(count != -1);

      System.out.println();
    } catch(InvalidPathException e) {
      System.out.println("Path Error " + e);
    } catch (IOException e) {
      System.out.println("I/O Error " + e);
    }
  }
}
```

In this version, the variable **filepath** is not needed and both exceptions are handled by the same **try** statement. Because this approach is more compact, it is the approach used in the rest of the examples in this chapter. Of course, in your own code, you may encounter situations in which the creation of a **Path** object needs to be separate from the acquisition of a channel. In these cases, the previous approach can be used.

Another way to read a file is to map it to a buffer. The advantage is that the buffer automatically contains the contents of the file. No explicit read operation is necessary. To map and read the contents of a file, follow this general procedure. First, obtain a **Path** object that encapsulates the file as previously described. Next, obtain a channel to that file by calling **Files.newByteChannel( )**, passing in the **Path** and casting the returned object to **FileChannel**. As explained, **newByteChannel( )** returns a **SeekableByteChannel**. When using the default file system, this object can be cast to **FileChannel**. Then, map the channel to a buffer by calling **map( )** on the channel. The **map( )** method is defined by **FileChannel**. This is why the cast to **FileChannel** is needed. The **map( )** function is shown here:

```
MappedByteBuffer map(FileChannel.MapMode how,
    long pos, long size) throws IOException
```

The **map( )** method causes the data in the file to be mapped into a buffer in memory. The value in *how* determines what type of operations are allowed. It must be one of these values:

MapMode.READ_ONLY	MapMode.READ_WRITE	MapMode.PRIVATE

For reading a file, use **MapMode.READ_ONLY**. To read and write, use **MapMode.READ_WRITE**. **MapMode.PRIVATE** causes a private copy of the file to be made, and changes to the buffer do not affect the underlying file. The location within the file to begin mapping is specified by *pos*, and the number of bytes to map are specified by *size*. A reference to this buffer is returned as a **MappedByteBuffer**, which is a subclass of **ByteBuffer**. Once the file has been mapped to a buffer, you can read the file from that buffer. Here is an example that illustrates this approach:

```
// Use a mapped file to read a file.

import java.io.*;
import java.nio.*;
import java.nio.channels.*;
import java.nio.file.*;

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

    // Obtain a channel to a file within a try-with-resources block.
    try ( FileChannel fChan =
         (FileChannel) Files.newByteChannel(Path.of("test.txt")) )
    {

      // Get the size of the file.
      long fSize = fChan.size();

      // Now, map the file into a buffer.
      MappedByteBuffer mBuf = fChan.map(FileChannel.MapMode.READ_ONLY, 0, fSize);

      // Read and display bytes from buffer.
      for(int i=0; i < fSize; i++)
        System.out.print((char)mBuf.get());

      System.out.println();

    } catch(InvalidPathException e) {
      System.out.println("Path Error " + e);
    } catch (IOException e) {
      System.out.println("I/O Error " + e);
    }
  }
}
```

In the program, a **Path** to the file is created and then opened via **newByteChannel( )**. The channel is cast to **FileChannel** and stored in **fChan**. Next, the size of the file is obtained by calling **size( )** on the channel. Then, the entire file is mapped into memory by calling **map( )** on **fChan** and a reference to the buffer is stored in **mBuf**. Notice that **mBuf** is declared as a reference to a **MappedByteBuffer**. The bytes in **mBuf** are read by calling **get( )**.

Writing to a File via a Channel

As is the case when reading from a file, there are also several ways to write data

to a file using a channel. We will begin with one of the most common. In this approach, you manually allocate a buffer, write data to that buffer, and then perform an explicit write operation to write that data to a file.

Before you can write to a file, you must open it. To do this, first obtain a **Path** that describes the file and then use this **Path** to open the file. In this example, the file will be opened for byte-based output via explicit output operations. Therefore, this example will open the file and establish a channel to it by calling **Files.newByteChannel( )**. As shown in the previous section, the **newByteChannel( )** method that we will use has this general form:

```
static SeekableByteChannel newByteChannel(Path path, OpenOption ...
    how)
    throws IOException
```

It returns a **SeekableByteChannel** object, which encapsulates the channel for file operations. To open a file for output, the *how* parameter must specify **StandardOpenOption.WRITE**. If you want to create the file if it does not already exist, then you must also specify **StandardOpenOption.CREATE**. (Other options, which are shown in Table 22-5, are also available.) As explained in the previous section, **SeekableByteChannel** is an interface that describes a channel that can be used for file operations. It is implemented by the **FileChannel** class. When the default file system is used, the return object can be cast to **FileChannel**. You must close the channel after you have finished with it.

Here is one way to write to a file through a channel using explicit calls to **write( )**. First, obtain a **Path** to the file and then open it with a call to **newByteChannel( )**, casting the result to **FileChannel**. Next, allocate a byte buffer and write data to that buffer. Before the data is written to the file, call **rewind( )** on the buffer to set its current position to zero. (Each output operation on the buffer increases the current position. Thus, it must be reset prior to writing to the file.) Then, call **write( )** on the channel, passing in the buffer. The following program demonstrates this procedure. It writes the alphabet to a file called **test.txt**.

```
// Write to a file using NIO.

import java.io.*;
import java.nio.*;
import java.nio.channels.*;
import java.nio.file.*;

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

    // Obtain a channel to a file within a try-with-resources block.
    try ( FileChannel fChan = (FileChannel)
            Files.newByteChannel(Path.of("test.txt"),
                                 StandardOpenOption.WRITE,
                                 StandardOpenOption.CREATE) )
    {
      // Create a buffer.
      ByteBuffer mBuf = ByteBuffer.allocate(26);

      // Write some bytes to the buffer.
      for(int i=0; i<26; i++)
        mBuf.put((byte)('A' + i));

      // Reset the buffer so that it can be written.
      mBuf.rewind();

      // Write the buffer to the output file.
      fChan.write(mBuf);

    } catch(InvalidPathException e) {
      System.out.println("Path Error " + e);
    } catch (IOException e) {
      System.out.println("I/O Error: " + e);
      System.exit(1);
    }
  }
}
```

It is useful to emphasize an important aspect of this program. As mentioned, after data is written to **mBuf**, but before it is written to the file, a call to **rewind( )** on **mBuf** is made. This is necessary in order to reset the current position to zero after data has been written to **mBuf**. Remember, each call to **put( )** on **mBuf** advances the current position. Therefore, it is necessary for the current position

to be reset to the start of the buffer before calling **write( )**. If this is not done, **write( )** will think that there is no data in the buffer.

Another way to handle the resetting of the buffer between input and output operations is to call **flip( )** instead of **rewind( )**. The **flip( )** method sets the value of the current position to zero and the limit to the previous current position. In the preceding example, because the capacity of the buffer equals its limit, **flip( )** could have been used instead of **rewind( )**. However, the two methods are not interchangeable in all cases.

In general, you must reset the buffer between read and write operations. For example, assuming the preceding example, the following loop will write the alphabet to the file three times. Pay special attention to the calls to **rewind( )**.

```
for(int h=0; h<3; h++) {
  // Write some bytes to the buffer.
  for(int i=0; i<26; i++)
    mBuf.put((byte)('A' + i));

  // Rewind the buffer so that it can be written.
  mBuf.rewind();

  // Write the buffer to the output file.
  fChan.write(mBuf);

  // Rewind the buffer so that it can be written to again.
  mBuf.rewind();
}
```

Notice that **rewind( )** is called between each read and write operation.

One other thing about the program warrants mentioning: When the buffer is written to the file, the first 26 bytes in the file will contain the output. If the file **test.txt** was preexisting, then after the program executes, the first 26 bytes of **test.txt** will contain the alphabet, but the remainder of the file will remain unchanged.

Another way to write to a file is to map it to a buffer. The advantage to this approach is that the data written to the buffer will automatically be written to the file. No explicit write operation is necessary. To map and write the contents of a file, we will use this general procedure. First, obtain a **Path** object that encapsulates the file and then create a channel to that file by calling **Files.newByteChannel( )**, passing in the **Path**. Cast the reference returned by **newByteChannel( )** to **FileChannel**. Next, map the channel to a buffer by

calling **map( )** on the channel. The **map( )** method was described in detail in the previous section. It is summarized here for your convenience. Here is its general form:

```
MappedByteBuffer map(FileChannel.MapMode how,
    long pos, long size) throws IOException
```

The **map( )** method causes the data in the file to be mapped into a buffer in memory. The value in *how* determines what type of operations are allowed. For writing to a file, *how* must be **MapMode.READ_WRITE**. The location within the file to begin mapping is specified by *pos*, and the number of bytes to map are specified by *size*. A reference to this buffer is returned. Once the file has been mapped to a buffer, you can write data to that buffer, and it will automatically be written to the file. Therefore, no explicit write operations to the channel are necessary.

Here is the preceding program reworked so that a mapped file is used. Notice that in the call to **newByteChannel( )**, the open option **StandardOpenOption.READ** has been added. This is because a mapped buffer can either be read-only or read/write. Thus, to write to the mapped buffer, the channel must be opened as read/write.

```
// Write to a mapped file.

import java.io.*;
import java.nio.*;
import java.nio.channels.*;
import java.nio.file.*;

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

    // Obtain a channel to a file within a try-with-resources block.
    try ( FileChannel fChan = (FileChannel)
          Files.newByteChannel(Path.of("test.txt"),
                   StandardOpenOption.WRITE,
                   StandardOpenOption.READ,
                   StandardOpenOption.CREATE) )
    {

      // Then, map the file into a buffer.
      MappedByteBuffer mBuf = fChan.map(FileChannel.MapMode.READ_WRITE, 0, 26);

      // Write some bytes to the buffer.
      for(int i=0; i<26; i++)
        mBuf.put((byte)('A' + i));

    } catch(InvalidPathException e) {
      System.out.println("Path Error " + e);

    } catch (IOException e) {
      System.out.println("I/O Error " + e);
    }
  }
}
```

As you can see, there are no explicit write operations to the channel itself. Because **mBuf** is mapped to the file, changes to **mBuf** are automatically reflected in the underlying file.

Copying a File Using NIO

NIO simplifies several types of file operations. Although we can't examine them all, an example will give you an idea of what is available. The following program copies a file using a call to a single NIO method: **copy( )**, which is a **static** method defined by **Files**. It has several forms. Here is the one we will be

using:

static Path copy(Path *src*, Path *dest*, CopyOption ... *how*) throws IOException

The file specified by *src* is copied to the file specified by *dest*. How the copy is performed is specified by *how*. Because it is a varargs parameter, it can be missing. If specified, it can be one or more of these values, which are valid for all file systems:

StandardCopyOption.COPY_ATTRIBUTES	Request that the file's attributes be copied.
StandardLinkOption.NOFOLLOW_LINKS	Do not follow symbolic links.
StandardCopyOption.REPLACE_EXISTING	Overwrite a preexisting file.

Other options may be supported, depending on the implementation.

The following program demonstrates **copy( )**. The source and destination files are specified on the command line, with the source file specified first. Notice how short the program is. You might want to compare this version of the file copy program to the one found in Chapter 13. As you will find, the part of the program that actually copies the file is substantially shorter in the NIO version shown here.

```
// Copy a file using NIO.
import java.io.*;
import java.nio.*;
import java.nio.channels.*;
import java.nio.file.*;

public class NIOCopy {

  public static void main(String args[]) {

    if(args.length != 2) {
      System.out.println("Usage: Copy from to");
      return;
    }

    try {
      Path source = Path.of(args[0]);
      Path target = Path.of(args[1]);

      // Copy the file.
      Files.copy(source, target, StandardCopyOption.REPLACE_EXISTING);

    } catch(InvalidPathException e) {
      System.out.println("Path Error " + e);
    } catch (IOException e) {
      System.out.println("I/O Error " + e);
    }
  }
}
```

Use NIO for Stream-Based I/O

Beginning with NIO.2, you can use NIO to open an I/O stream. Once you have a **Path**, open a file by calling **newInputStream( )** or **newOutputStream( )**, which are **static** methods defined by **Files**. These methods return a stream connected to the specified file. In either case, the stream can then be operated on in the way described in Chapter 20, and the same techniques apply. The advantage of using **Path** to open a file is that all of the features defined by NIO are available for your use.

To open a file for stream-based input, use **Files.newInputStream( )**. It is shown here:

static InputStream newInputStream(Path *path*, OpenOption ... *how*)
    throws IOException

Here, *path* specifies the file to open and *how* specifies how the file will be opened. It must be one or more of the values defined by **StandardOpenOption**, described earlier. (Of course, only those options that relate to an input stream will apply.) If no options are specified, then the file is opened as if **StandardOpenOption.READ** were passed.

Once opened, you can use any of the methods defined by **InputStream**. For example, you can use **read( )** to read bytes from the file.

The following program demonstrates the use of NIO-based stream I/O. It reworks the **ShowFile** program from Chapter 13 so that it uses NIO features to open the file and obtain a stream. As you can see, it is very similar to the original, except for the use of **Path** and **newInputStream( )**.

```
/* Display a text file using stream-based, NIO code.
   To use this program, specify the name
   of the file that you want to see.
   For example, to see a file called TEST.TXT,
   use the following command line.

   java ShowFile TEST.TXT
*/

import java.io.*;
import java.nio.file.*;

class ShowFile {
  public static void main(String args[])
  {
    int i;

    // First, confirm that a filename has been specified.
    if(args.length != 1) {
      System.out.println("Usage: ShowFile filename");
      return;
    }

    // Open the file and obtain a stream linked to it.
    try ( InputStream fin = Files.newInputStream(Path.of(args[0])) )
    {
      do {
        i = fin.read();
        if(i != -1) System.out.print((char) i);
      } while(i != -1);

    } catch(InvalidPathException e) {
      System.out.println("Path Error " + e);
    } catch(IOException e) {
      System.out.println("I/O Error "  + e);
    }
  }
}
```

Because the stream returned by **newInputStream( )** is a normal stream, it can be used like any other stream. For example, you can wrap the stream inside a buffered stream, such as a **BufferedInputStream**, to provide buffering, as shown here:

```
new BufferedInputStream(Files.newInputStream(Path.of(args[0])))
```

Now, all reads will be automatically buffered.

To open a file for output, use **Files.newOutputStream( )**. It is shown here:

static OutputStream newOutputStream(Path *path*, OpenOption ... *how*)
throws IOException

Here, *path* specifies the file to open and *how* specifies how the file will be opened. It must be one or more of the values defined by **StandardOpenOption**, described earlier. (Of course, only those options that relate to an output stream will apply.) If no options are specified, then the file is opened as if **StandardOpenOption.WRITE**, **StandardOpenOption.CREATE**, and **StandardOpenOption.TRUNCATE_EXISTING** were passed.

The methodology for using **newOutputStream( )** is similar to that shown previously for **newInputStream( )**. Once opened, you can use any of the methods defined by **OutputStream**. For example, you can use **write( )** to write bytes to the file. You can also wrap the stream inside a **BufferedOutputStream** to buffer the stream.

The following program shows **newOutputStream( )** in action. It writes the alphabet to a file called **test.txt**. Notice the use of buffered I/O.

```
// Demonstrate NIO-based, stream output.

import java.io.*;
import java.nio.file.*;

class NIOStreamWrite {
  public static void main(String args[])
  {
    // Open the file and obtain a stream linked to it.
    try ( OutputStream fout =
          new BufferedOutputStream(
                Files.newOutputStream(Path.of("test.txt"))) )
    {
      // Write some bytes to the stream.
      for(int i=0; i < 26; i++)
        fout.write((byte)('A' + i));

    } catch(InvalidPathException e) {
      System.out.println("Path Error " + e);
    } catch(IOException e) {
      System.out.println("I/O Error: " + e);
    }
  }
}
```

Use NIO for Path and File System Operations

At the beginning of Chapter 21, the **File** class in the **java.io** package was examined. As explained there, the **File** class deals with the file system and with the various attributes associated with a file, such as whether a file is read-only, hidden, and so on. It was also used to obtain information about a file's path. Although the **File** class is still perfectly acceptable, the interfaces and classes defined by NIO.2 offer a better way to perform these functions. The benefits include support for symbolic links, better support for directory tree traversal, and improved handling of metadata, among others. The following sections show samples of two common file system operations: obtaining information about a path and file and getting the contents of a directory.

REMEMBER If you want to change code that uses **java.io.File** to the **Path** interface, you can use the **toPath( )** method to obtain a **Path** instance from a **File** instance.

Obtain Information About a Path and a File

Information about a path can be obtained by using methods defined by **Path**. Some attributes associated with the file described by a **Path** (such as whether or not the file is hidden) are obtained by using methods defined by **Files**. The **Path** methods used here are **getName( )**, **getParent( )**, and **toAbsolutePath( )**. Those provided by **Files** are **isExecutable( )**, **isHidden( )**, **isReadable( )**, **isWritable( )**, and **exists( )**. These are summarized in Tables 22-3 and 22-4, shown earlier.

CAUTION Methods such as **isExecutable( ), isReadable( ), isWritable( ),** and **exists( )** must be used with care because the state of the file system may change after the call, in which case a program malfunction could occur. Such a situation could have security implications.

Other file attributes are obtained by requesting a list of attributes by calling **Files.readAttributes( )**. In the program, this method is called to obtain the **BasicFileAttributes** associated with a file, but the general approach applies to other types of attributes.

The following program demonstrates several of the **Path** and **Files** methods, along with several methods provided by **BasicFileAttributes**. This program assumes that a file called **test.txt** exists in a directory called **examples**, which must be a subdirectory of the current directory.

```
// Obtain information about a path and a file.
import java.io.*;
import java.nio.file.*;
import java.nio.file.attribute.*;

class PathDemo {
  public static void main(String args[]) {
    Path filepath = Path.of("examples\\test.txt");

    System.out.println("File Name: " + filepath.getName(1));
    System.out.println("Path: " + filepath);
    System.out.println("Absolute Path: " + filepath.toAbsolutePath());
    System.out.println("Parent: " + filepath.getParent());

    if(Files.exists(filepath))
      System.out.println("File exists");
    else
      System.out.println("File does not exist");

    try {
      if(Files.isHidden(filepath))
        System.out.println("File is hidden");
      else
        System.out.println("File is not hidden");
    } catch(IOException e) {
        System.out.println("I/O Error: " + e);
    }

    Files.isWritable(filepath);
    System.out.println("File is writable");


    Files.isReadable(filepath);
    System.out.println("File is readable");

    try {
      BasicFileAttributes attribs =
        Files.readAttributes(filepath, BasicFileAttributes.class);

      if(attribs.isDirectory())
        System.out.println("The file is a directory");
      else
        System.out.println("The file is not a directory");

      if(attribs.isRegularFile())
        System.out.println("The file is a normal file");

      else
        System.out.println("The file is not a normal file");

      if(attribs.isSymbolicLink())
        System.out.println("The file is a symbolic link");
      else
        System.out.println("The file is not a symbolic link");

      System.out.println("File last modified: " + attribs.lastModifiedTime());
      System.out.println("File size: " + attribs.size() + " Bytes");
    } catch(IOException e) {
      System.out.println("Error reading attributes: " + e);
    }
  }
}
```

If you execute this program from a directory called **MyDir**, which has a subdirectory called **examples**, and the **examples** directory contains the **test.txt** file, then you will see output similar to that shown here. (Of course, the information you see will differ.)

```
File Name: test.txt
Path: examples\test.txt
Absolute Path: C:\MyDir\examples\test.txt
Parent: examples
File exists
File is not hidden
File is writable
File is readable
The file is not a directory
The file is a normal file
The file is not a symbolic link
File last modified: 2017-01-01T18:20:46.380445Z
File size: 18 Bytes
```

If you are using a computer that supports the FAT file system (i.e., the DOS file system), then you might want to try using the methods defined by **DosFileAttributes**. If you are using a POSIX-compatible system, then try using **PosixFileAttributes**.

List the Contents of a Directory

If a path describes a directory, then you can read the contents of that directory by using **static** methods defined by **Files**. To do this, you first obtain a directory stream by calling **newDirectoryStream( )**, passing in a **Path** that describes the directory. One form of **newDirectoryStream( )** is shown here:

static DirectoryStream<Path> newDirectoryStream(Path *dirPath*)
    throws IOException

Here, *dirPath* encapsulates the path to the directory. The method returns a **DirectoryStream<Path>** object that can be used to obtain the contents of the directory. It will throw an **IOException** if an I/O error occurs and a **NotDirectoryException** (which is a subclass of **IOException**) if the specified path is not a directory. A **SecurityException** is also possible if access to the directory is not permitted.

DirectoryStream<Path> implements **AutoCloseable**, so it can be managed by a **try**-with-resources statement. It also implements **Iterable<Path>**. This

means that you can obtain the contents of the directory by iterating over the **DirectoryStream** object. When iterating, each directory entry is represented by a **Path** instance. An easy way to iterate over a **DirectoryStream** is to use a for-each style **for** loop. It is important to understand, however, that the iterator implemented by **DirectoryStream<Path>** can be obtained only once for each instance. Thus, the **iterator( )** method can be called only once, and a for-each loop can be executed only once.

The following program displays the contents of a directory called **MyDir**:

```
// Display a directory.

import java.io.*;
import java.nio.file.*;
import java.nio.file.attribute.*;

class DirList {
  public static void main(String args[]) {
    String dirname = "\\MyDir";

    // Obtain and manage a directory stream within a try block.
    try ( DirectoryStream<Path> dirstrm =
            Files.newDirectoryStream(Path.of(dirname)) )
    {
      System.out.println("Directory of " + dirname);

      // Because DirectoryStream implements Iterable, we
      // can use a "foreach" loop to display the directory.
      for(Path entry : dirstrm) {
        BasicFileAttributes attribs =
             Files.readAttributes(entry, BasicFileAttributes.class);

        if(attribs.isDirectory())
          System.out.print("<DIR> ");
        else
          System.out.print("      ");

        System.out.println(entry.getName(1));
      }
    } catch(InvalidPathException e) {
      System.out.println("Path Error " + e);
    } catch(NotDirectoryException e) {
      System.out.println(dirname + " is not a directory.");
    } catch (IOException e) {
      System.out.println("I/O Error: " + e);
    }
  }
}
```

Here is sample output from the program:

```
Directory of \MyDir
      DirList.class
      DirList.java
<DIR> examples
      Test.txt
```

You can filter the contents of a directory in two ways. The easiest is to use this version of **newDirectoryStream( )**:

static DirectoryStream<Path> newDirectoryStream(Path *dirPath*, String *wildcard*)
throws IOException

In this version, only files that match the wildcard filename specified by *wildcard* will be obtained. For *wildcard*, you can specify either a complete filename or a *glob*. A *glob* is a string that defines a general pattern that will match one or more files using the familiar ***** and **?** wildcard characters. These match zero or more of any character and any one character, respectively. The following are also recognized within a glob:

**	Matches zero or more of any character across directories.
[*chars*]	Matches any one character in *chars*. A * or ? within *chars* will be treated as a normal character, not a wildcard. A range can be specified by use of a hyphen, such as [x-z].
{*globlist*}	Matches any one of the globs specified in a comma-separated list of globs in *globlist*.

You can specify a ***** or **?** character, using ***** and **\?**. To specify a \, use \\. You can experiment with a glob by substituting this call to **newDirectoryStream( )** into the previous program:

```
Files.newDirectoryStream(Path.of(dirname), "{Path,Dir}*.
{java,class}")
```

This obtains a directory stream that contains only those files whose names begin with either "Path" or "Dir" and use either the "java" or "class" extension. Thus, it would match names like **DirList.java** and **PathDemo.java**, but not **MyPathDemo.java**, for example.

Another way to filter a directory is to use this version of **newDirectoryStream( )**:

static DirectoryStream<Path> newDirectoryStream(Path *dirPath*,
DirectoryStream.Filter<? super Path> *filefilter*)

throws IOException

Here, **DirectoryStream.Filter** is an interface that specifies the following method:

boolean accept(T *entry*) throws IOException

In this case, **T** will be **Path**. If you want to include *entry* in the list, return **true**. Otherwise, return **false**. This form of **newDirectoryStream( )** offers the advantage of being able to filter a directory based on something other than a filename. For example, you can filter based on size, creation date, modification date, or attribute, to name a few.

The following program demonstrates the process. It will list only those files that are writable.

```
// Display a directory of only those files that are writable.

import java.io.*;
import java.nio.file.*;
import java.nio.file.attribute.*;

class DirList {
  public static void main(String args[]) {
    String dirname = "\\MyDir";

    // Create a filter that returns true only for writable files.
    DirectoryStream.Filter<Path> how = new DirectoryStream.Filter<Path>() {
      public boolean accept(Path filename) throws IOException {
        if(Files.isWritable(filename)) return true;
        return false;
      }
    };

    // Obtain and manage a directory stream of writable files.
    try (DirectoryStream<Path> dirstrm =
           Files.newDirectoryStream(Path.of(dirname), how) )
    {
      System.out.println("Directory of " + dirname);

      for(Path entry : dirstrm) {
        BasicFileAttributes attribs =
          Files.readAttributes(entry, BasicFileAttributes.class);

        if(attribs.isDirectory())
          System.out.print("<DIR> ");
        else
          System.out.print("      ");

        System.out.println(entry.getName(1));
      }
    } catch(InvalidPathException e) {
      System.out.println("Path Error " + e);
    } catch(NotDirectoryException e) {
      System.out.println(dirname + " is not a directory.");
    } catch (IOException e) {
      System.out.println("I/O Error: " + e);
    }
  }
}
```

Use walkFileTree( ) to List a Directory Tree

The preceding examples have obtained the contents of only a single directory. However, sometimes you will want to obtain a list of the files in a directory tree. In the past, this was quite a chore, but NIO.2 makes it easy because now you can use the **walkFileTree( )** method defined by **Files** to process a directory tree. It has two forms. The one used in this chapter is shown here:

```
static Path walkFileTree(Path root, FileVisitor<? super Path> fv)
    throws IOException
```

The path to the starting point of the directory walk is passed in *root*. An instance of **FileVisitor** is passed in *fv*. The implementation of **FileVisitor** determines how the directory tree is traversed, and it gives you access to the directory information. If an I/O error occurs, an **IOException** is thrown. A **SecurityException** is also possible.

FileVisitor is an interface that defines how files are visited when a directory tree is traversed. It is a generic interface that is declared like this:

```
interface FileVisitor<T>
```

For use in **walkFileTree( )**, **T** will be **Path** (or any type derived from **Path**). **FileVisitor** defines the following methods:

Method	Description
FileVisitResult postVisitDirectory(T *dir*, IOException *exc*) throws IOException	Called after a directory has been visited. The directory is passed in *dir*, and any **IOException** is passed in *exc*. If *exc* is **null**, no exception occurred. The result is returned.
FileVisitResult preVisitDirectory(T *dir*, BasicFileAttributes *attribs*) throws IOException	Called before a directory is visited. The directory is passed in *dir*, and the attributes associated with the directory are passed in *attribs*. The result is returned. To examine the directory, return **FileVisitResult.CONTINUE**.
FileVisitResult visitFile(T *file*, BasicFileAttributes *attribs*) throws IOException	Called when a file is visited. The file is passed in *file*, and the attributes associated with the file are passed in *attribs*. The result is returned.
FileVisitResult visitFileFailed(T *file*, IOException *exc*) throws IOException	Called when an attempt to visit a file fails. The file that failed is passed in *file*, and the **IOException** is passed in *exc*. The result is returned.

Notice that each method returns a **FileVisitResult**. This enumeration defines the

following values:

CONTINUE	SKIP_SIBLINGS	SKIP_SUBTREE	TERMINATE

In general, to continue traversing the directory and subdirectories, a method should return **CONTINUE**. For **preVisitDirectory( )**, return **SKIP_SIBLINGS** to bypass the directory and its siblings and prevent **postVisitDirectory( )** from being called. To bypass just the directory and subdirectories, return **SKIP_SUBTREE**. To stop the directory traversal, return **TERMINATE**.

Although it is certainly possible to create your own visitor class that implements these methods defined by **FileVisitor**, you won't normally do so because a simple implementation is provided by **SimpleFileVisitor**. You can just override the default implementation of the method or methods in which you are interested. Here is a short example that illustrates the process. It displays all files in the directory tree that has **MyDir** as its root. Notice how short this program is.

```
// A simple example that uses walkFileTree( ) to display a directory tree.
import java.io.*;
import java.nio.file.*;
import java.nio.file.attribute.*;


// Create a custom version of SimpleFileVisitor that overrides
// the visitFile( ) method.
class MyFileVisitor extends SimpleFileVisitor<Path> {
  public FileVisitResult visitFile(Path path, BasicFileAttributes attribs)
    throws IOException
  {
    System.out.println(path);
    return FileVisitResult.CONTINUE;
  }
}

class DirTreeList {
  public static void main(String args[]) {
    String dirname = "\\MyDir";

    System.out.println("Directory tree starting with " + dirname + ":\n");

    try {
      Files.walkFileTree(Path.of(dirname), new MyFileVisitor());
    } catch (IOException exc) {
      System.out.println("I/O Error");
    }
  }
}
```

Here is sample output produced by the program when used on the same **MyDir** directory shown earlier. In this example, the subdirectory called **examples** contains one file called **MyProgram.java**.

```
Directory tree starting with \MyDir:

\MyDir\DirList.class
\MyDir\DirList.java
\MyDir\examples\MyProgram.java
\MyDir\Test.txt
```

In the program, the class **MyFileVisitor** extends **SimpleFileVisitor**, overriding only the **visitFile( )** method. In this example, **visitFile( )** simply displays the

files, but more sophisticated functionality is easy to achieve. For example, you could filter the files or perform actions on the files, such as copying them to a backup device. For the sake of clarity, a named class was used to override **visitFile( )**, but you could also use an anonymous inner class.

One last point: It is possible to watch a directory for changes by using **java.nio.file.WatchService**.

CHAPTER

23

Networking

Since its beginning, Java has been associated with Internet programming. There are a number of reasons for this, not the least of which is its ability to generate secure, cross-platform, portable code. However, one of the most important reasons that Java became the premier language for network programming are the classes defined in the **java.net** package. They provide a convenient means by which programmers of all skill levels can access network resources. Beginning with JDK 11, Java has also provided enhanced networking support for HTTP clients in the **java.net.http** package in a module by the same name. Called the HTTP Client API, it further solidifies Java's networking capabilities.

This chapter explores the **java.net** package. It concludes by introducing the new **java.http**
.net package. It is important to emphasize that networking is a very large and at times complicated topic. It is not possible for this book to discuss all of the capabilities contained in these two packages. Instead, this chapter focuses on several of their core classes and interfaces.

Networking Basics

Before we begin, it will be useful to review some key networking concepts and terms. At the core of Java's networking support is the concept of a *socket*. A socket identifies an endpoint in a network. The socket paradigm was part of the 4.2BSD Berkeley UNIX release in the early 1980s. Because of this, the term *Berkeley socket* is also used. Sockets are at the foundation of modern networking because a socket allows a single computer to serve many different clients at once, as well as to serve many different types of information. This is accomplished through the use of a *port*, which is a numbered socket on a particular machine. A server process is said to "listen" to a port until a client connects to it. A server is allowed to accept multiple clients connected to the same port number, although each session is unique. To manage multiple client connections, a server process must be multithreaded or have some other means of multiplexing the simultaneous I/O.

Socket communication takes place via a protocol. *Internet Protocol (IP)* is a

low-level routing protocol that breaks data into small packets and sends them to an address across a network, which does not guarantee to deliver said packets to the destination. *Transmission Control Protocol (TCP)* is a higher-level protocol that manages to robustly string together these packets, sorting and retransmitting them as necessary to reliably transmit data. A third protocol, *User Datagram Protocol (UDP),* sits next to TCP and can be used directly to support fast, connectionless, unreliable transport of packets.

Once a connection has been established, a higher-level protocol ensues, which is dependent on which port you are using. TCP/IP reserves the lower 1,024 ports for specific protocols. Many of these will seem familiar to you if you have spent any time surfing the Internet. Port number 21 is for FTP; 23 is for Telnet; 25 is for e-mail; 43 is for whois; 80 is for HTTP; 119 is for netnews—and the list goes on. It is up to each protocol to determine how a client should interact with the port.

For example, HTTP is the protocol that web browsers and servers use to transfer hypertext pages and images. It is a quite simple protocol for a basic page-browsing web server. Here's how it works. When a client requests a file from an HTTP server, an action known as a *hit,* it simply sends the name of the file in a special format to a predefined port and reads back the contents of the file. The server also responds with a status code to tell the client whether or not the request can be fulfilled and why.

A key component of the Internet is the *address*. Every computer on the Internet has one. An Internet address is a number that uniquely identifies each computer on the Net. Originally, all Internet addresses consisted of 32-bit values, organized as four 8-bit values. This address type was specified by IPv4 (Internet Protocol, version 4). However, a new addressing scheme, called IPv6 (Internet Protocol, version 6) has come into play. IPv6 uses a 128-bit value to represent an address, organized into eight 16-bit chunks. Although there are several reasons for and advantages to IPv6, the main one is that it supports a much larger address space than does IPv4. Fortunately, when using Java, you won't normally need to worry about whether IPv4 or IPv6 addresses are used because Java handles the details for you.

Just as the numbers of an IP address describe a network hierarchy, the name of an Internet address, called its *domain name,* describes a machine's location in a name space. For example, **www.HerbSchildt.com** is in the *COM* top-level domain (used by U.S. commercial sites); it is called *HerbSchildt*, and *www* identifies the server for web requests. An Internet domain name is mapped to an IP address by the *Domain Naming Service (DNS).* This enables users to work

with domain names, but the Internet operates on IP addresses.

The java.net Networking Classes and Interfaces

The **java.net** package contains Java's original networking features, which have been available since version 1.0. It supports TCP/IP both by extending the already established stream I/O interface introduced in Chapter 21 and by adding the features required to build I/O objects across the network. Java supports both the TCP and UDP protocol families. TCP is used for reliable stream-based I/O across the network. UDP supports a simpler, hence faster, point-to-point datagram-oriented model. The classes contained in the **java.net** package are shown here:

Authenticator	InetAddress	SocketAddress
CacheRequest	InetSocketAddress	SocketImpl
CacheResponse	InterfaceAddress	SocketPermission
ContentHandler	JarURLConnection	StandardSocketOption
CookieHandler	MulticastSocket	URI
CookieManager	NetPermission	URL
DatagramPacket	NetworkInterface	URLClassLoader
DatagramSocket	PasswordAuthentication	URLConnection
DatagramSocketImpl	Proxy	URLDecoder
HttpCookie	ProxySelector	URLEncoder
HttpURLConnection	ResponseCache	URLPermission
IDN	SecureCacheResponse	URLStreamHandler
Inet4Address	ServerSocket	
Inet6Address	Socket	

The **java.net** package's interfaces are listed here:

ContentHandlerFactory	FileNameMap	SocketOptions
CookiePolicy	ProtocolFamily	URLStreamHandlerFactory
CookieStore	SocketImplFactory	
DatagramSocketImplFactory	SocketOption	

Beginning with JDK 9, **java.net** is part of the **java.base** module. In the sections that follow, we will examine the main networking classes and show several examples that apply to them. Once you understand these core networking classes, you will be able to easily explore the others on your own.

InetAddress

The **InetAddress** class is used to encapsulate both the numerical IP address and the domain name for that address. You interact with this class by using the name of an IP host, which is more convenient and understandable than its IP address. The **InetAddress** class hides the number inside. **InetAddress** can handle both IPv4 and IPv6 addresses.

Factory Methods

The **InetAddress** class has no visible constructors. To create an **InetAddress** object, you have to use one of the available factory methods. As explained earlier in this book, *factory methods* are merely a convention whereby static methods in a class return an instance of that class. This is done in lieu of overloading a constructor with various parameter lists when having unique method names makes the results much clearer. Three commonly used **InetAddress** factory methods are shown here:

```
static InetAddress getLocalHost( )
    throws UnknownHostException
static InetAddress getByName(String hostName)
    throws UnknownHostException
static InetAddress[ ] getAllByName(String hostName)
    throws UnknownHostException
```

The **getLocalHost( )** method simply returns the **InetAddress** object that represents the local host. The **getByName( )** method returns an **InetAddress** for a host name passed to it. If these methods are unable to resolve the host name,

they throw an **UnknownHostException**.

On the Internet, it is common for a single name to be used to represent several machines. In the world of web servers, this is one way to provide some degree of scaling. The **getAllByName( )** factory method returns an array of **InetAddress**es that represent all of the addresses that a particular name resolves to. It will also throw an **UnknownHostException** if it can't resolve the name to at least one address.

InetAddress also includes the factory method **getByAddress( ),** which takes an IP address and returns an **InetAddress** object. Either an IPv4 or an IPv6 address can be used.

The following example prints the addresses and names of the local machine and two Internet web sites:

```
// Demonstrate InetAddress.
import java.net.*;

class InetAddressTest
{
  public static void main(String args[]) throws UnknownHostException {
    InetAddress Address = InetAddress.getLocalHost();
    System.out.println(Address);

    Address = InetAddress.getByName("www.HerbSchildt.com");
    System.out.println(Address);

    InetAddress SW[] = InetAddress.getAllByName("www.nba.com");
    for (int i=0; i<SW.length; i++)
      System.out.println(SW[i]);
  }
}
```

Here is the output produced by this program. (Of course, the output you see may be slightly different.)

```
default/166.203.115.212
www.HerbSchildt.com/216.92.65.4
www.nba.com/23.61.252.147
www.nba.com/2600:1403:1:58d:0:0:0:2e1
www.nba.com/2600:1403:1:593:0:0:0:2e1
```

Instance Methods

The **InetAddress** class has several other methods, which can be used on the objects returned by the methods just discussed. Here is a sampling:

boolean equals(Object *other*)	Returns **true** if this object has the same Internet address as *other*.
byte[ ] getAddress( )	Returns a byte array that represents the object's IP address in network byte order.
String getHostAddress( )	Returns a string that represents the host address associated with the **InetAddress** object.
String getHostName( )	Returns a string that represents the host name associated with the **InetAddress** object.
boolean isMulticastAddress( )	Returns **true** if this address is a multicast address. Otherwise, it returns **false.**
String toString( )	Returns a string that lists the host name and the IP address for convenience.

Internet addresses are looked up in a series of hierarchically cached servers. That means that your local computer might know a particular name-to-IP-address mapping automatically, such as for itself and nearby servers. For other names, it may ask a local DNS server for IP address information. If that server doesn't have a particular address, it can go to a remote site and ask for it. This can continue all the way up to the root server. This process might take a long time, so it is wise to structure your code so that you cache IP address information locally rather than look it up repeatedly.

Inet4Address and Inet6Address

Java includes support for both IPv4 and IPv6 addresses. Because of this, two subclasses of **InetAddress** were created: **Inet4Address** and **Inet6Address**. **Inet4Address** represents a traditional-style IPv4 address. **Inet6Address** encapsulates a newer IPv6 address. Because they are subclasses of **InetAddress,** an **InetAddress** reference can refer to either. This is one way that Java was able to add IPv6 functionality without breaking existing code or adding many more classes. For the most part, you can simply use **InetAddress** when working with IP addresses because it can accommodate both styles.

TCP/IP Client Sockets

TCP/IP sockets are used to implement reliable, bidirectional, persistent, point-to-point, stream-based connections between hosts on the Internet. A socket can be used to connect Java's I/O system to other programs that may reside either on the local machine or on any other machine on the Internet, subject to security constraints.

There are two kinds of TCP sockets in Java. One is for servers, and the other is for clients. The **ServerSocket** class is designed to be a "listener," which waits for clients to connect before doing anything. Thus, **ServerSocket** is for servers. The **Socket** class is for clients. It is designed to connect to server sockets and initiate protocol exchanges. Because client sockets are the most commonly used by Java applications, they are examined here.

The creation of a **Socket** object implicitly establishes a connection between the client and server. There are no methods or constructors that explicitly expose the details of establishing that connection. Here are two constructors used to create client sockets:

Socket(String *hostName,* int *port*) throws UnknownHostException, IOException	Creates a socket connected to the named host and port.
Socket(InetAddress *ipAddress,* int *port*) throws IOException	Creates a socket using a preexisting **InetAddress** object and a port.

Socket defines several instance methods. For example, a **Socket** can be examined at any time for the address and port information associated with it, by use of the following methods:

InetAddress getInetAddress( )	Returns the **InetAddress** associated with the **Socket** object. It returns **null** if the socket is not connected.
int getPort( )	Returns the remote port to which the invoking **Socket** object is connected. It returns 0 if the socket is not connected.
int getLocalPort( )	Returns the local port to which the invoking **Socket** object is bound. It returns –1 if the socket is not bound.

You can gain access to the input and output streams associated with a **Socket** by use of the **getInputStream( )** and **getOuptutStream( )** methods, as shown

here. Each can throw an **IOException** if the socket has been invalidated by a loss of connection. These streams are used exactly like the I/O streams described in Chapter 21 to send and receive data.

InputStream getInputStream( ) throws IOException	Returns the **InputStream** associated with the invoking socket.
OutputStream getOutputStream( ) throws IOException	Returns the **OutputStream** associated with the invoking socket.

Several other methods are available, including **connect( ),** which allows you to specify a new connection; **isConnected( ),** which returns true if the socket is connected to a server; **isBound( ),** which returns true if the socket is bound to an address; and **isClosed( ),** which returns true if the socket is closed. To close a socket, call **close( )**. Closing a socket also closes the I/O streams associated with the socket. Beginning with JDK 7, **Socket** also implements **AutoCloseable**, which means that you can use a **try**-with-resources block to manage a socket.

The following program provides a simple **Socket** example. It opens a connection to a "whois" port (port 43) on the InterNIC server, sends the command-line argument down the socket, and then prints the data that is returned. InterNIC will try to look up the argument as a registered Internet domain name, and then send back the IP address and contact information for that site.

```
// Demonstrate Sockets.
import java.net.*;
import java.io.*;

class Whois {
  public static void main(String args[]) throws Exception {
    int c;

    // Create a socket connected to internic.net, port 43.
    Socket s = new Socket("whois.internic.net", 43);

    // Obtain input and output streams.
    InputStream in = s.getInputStream();
    OutputStream out = s.getOutputStream();

    // Construct a request string.

    String str = (args.length == 0 ? "OraclePressBooks.com" : args[0]) + "\n";
    // Convert to bytes.
    byte buf[] = str.getBytes();

    // Send request.
    out.write(buf);

    // Read and display response.
    while ((c = in.read()) != -1) {
      System.out.print((char) c);
    }
    s.close();
  }
}
```

Here is how the program works. First, a **Socket** is constructed that specifies the host name "whois.internic.net" and the port number 43. **Internic.net** is the InterNIC web site that handles whois requests. Port 43 is the whois port. Next, both input and output streams are opened on the socket. Then, a string is constructed that contains the name of the web site you want to obtain information about. In this case, if no web site is specified on the command line, then "OraclePressBooks.com" is used. The string is converted into a **byte** array and then sent out of the socket. The response is read by inputting from the socket, and the results are displayed. Finally, the socket is closed, which also closes the I/O streams.

In the preceding example, the socket was closed manually by calling **close( )**.

Beginning with JDK 7, you can use a **try**-with-resources block to automatically close the socket. For example, here is another way to write the **main( )** method of the previous program:

```
// Use try-with-resources to close a socket.
public static void main(String args[]) throws Exception {
  int c;

  // Create a socket connected to internic.net, port 43. Manage this
  // socket with a try-with-resources block.
  try ( Socket s = new Socket("whois.internic.net", 43) ) {

    // Obtain input and output streams.
    InputStream in = s.getInputStream();
    OutputStream out = s.getOutputStream();

    // Construct a request string.
    String str = (args.length == 0 ? "OraclePressBooks.com" : args[0]) + "\n";
    // Convert to bytes.
    byte buf[] = str.getBytes();

    // Send request.
    out.write(buf);

    // Read and display response.
    while ((c = in.read()) != -1) {
      System.out.print((char) c);
    }
  }
  // The socket is now closed.
}
```

In this version, the socket is automatically closed when the **try** block ends.

So the examples will work with earlier versions of Java and to clearly illustrate when a network resource can be closed, subsequent examples will continue to call **close( )** explicitly. However, in your own code, you should consider using automatic resource management since it offers a more streamlined approach. One other point: In this version, exceptions are still thrown out of **main( )**, but they could be handled by adding **catch** clauses to the end of the **try**-with-resources block.

NOTE For simplicity, the examples in this chapter simply throw all exceptions out of **main( )**. This allows the logic of the network code to be clearly illustrated. However, in real-world code, you will normally need to handle the exceptions in an appropriate way.

URL

The preceding example was rather obscure because the modern Internet is not about the older protocols such as whois, finger, and FTP. It is about WWW, the World Wide Web. The Web is a loose collection of higher-level protocols and file formats, all unified in a web browser. One of the most important aspects of the Web is that Tim Berners-Lee devised a scalable way to locate all of the resources of the Net. Once you can reliably name anything and everything, it becomes a very powerful paradigm. The Uniform Resource Locator (URL) does exactly that.

The URL provides a reasonably intelligible form to uniquely identify or address information on the Internet. URLs are ubiquitous; every browser uses them to identify information on the Web. Within Java's network class library, the **URL** class provides a simple, concise API to access information across the Internet using URLs.

All URLs share the same basic format, although some variation is allowed. Here are two examples: **http://www.HerbSchildt.com/** and **http://www.HerbSchildt.com:80/index.htm**. A URL specification is based on four components. The first is the protocol to use, separated from the rest of the locator by a colon (:). Common protocols are HTTP, FTP, gopher, and file, although these days almost everything is being done via HTTP (in fact, most browsers will proceed correctly if you leave off the "http://" from your URL specification). The second component is the host name or IP address of the host to use; this is delimited on the left by double slashes (//) and on the right by a slash (/) or optionally a colon (:). The third component, the port number, is an optional parameter, delimited on the left from the host name by a colon (:) and on the right by a slash (/). (It defaults to port 80, the predefined HTTP port; thus, ":80" is redundant.) The fourth part is the actual file path. Most HTTP servers will append a file named **index.html** or **index.htm** to URLs that refer directly to a directory resource. Thus, **http://www.HerbSchildt.com/** is the same as **http://www.HerbSchildt.com/index.htm**.

Java's **URL** class has several constructors; each can throw a **MalformedURLException**. One commonly used form specifies the URL with a string that is identical to what you see displayed in a browser:

URL(String *urlSpecifier*) throws MalformedURLException

The next two forms of the constructor allow you to break up the URL into its component parts:

URL(String *protocolName*, String *hostName*, int *port*, String *path*)
throws MalformedURLException

URL(String *protocolName*, String *hostName*, String *path*)
throws MalformedURLException

Another frequently used constructor allows you to use an existing URL as a reference context and then create a new URL from that context. Although this sounds a little contorted, it's really quite easy and useful.

URL(URL *urlObj*, String *urlSpecifier*) throws MalformedURLException

The following example creates a URL to a page on **HerbSchildt.com** and then examines its properties:

```
// Demonstrate URL.
import java.net.*;
class URLDemo {
  public static void main(String args[]) throws MalformedURLException {
    URL hp = new URL(http://www.HerbSchildt.com/WhatsNew");

    System.out.println("Protocol: " + hp.getProtocol());
    System.out.println("Port: " + hp.getPort());

    System.out.println("Host: " + hp.getHost());
    System.out.println("File: " + hp.getFile());
    System.out.println("Ext:" + hp.toExternalForm());
  }
}
```

When you run this, you will get the following output:

```
Protocol: http
Port: -1
Host: www.HerbSchildt.com
File: /WhatsNew
Ext:http://www.HerbSchildt.com/WhatsNew
```

Notice that the port is –1; this means that a port was not explicitly set. Given a **URL** object, you can retrieve the data associated with it. To access the actual bits or content information of a **URL,** create a **URLConnection** object from it, using its **openConnection( )** method, like this:

```
urlc = url.openConnection()
```

openConnection( ) has the following general form:

```
URLConnection openConnection( ) throws IOException
```

It returns a **URLConnection** object associated with the invoking **URL** object. Notice that it may throw an **IOException**.

URLConnection

URLConnection is a general-purpose class for accessing the attributes of a remote resource. Once you make a connection to a remote server, you can use **URLConnection** to inspect the properties of the remote object before actually transporting it locally. These attributes are exposed by the HTTP protocol specification and, as such, only make sense for **URL** objects that are using the HTTP protocol.

URLConnection defines several methods. Here is a sampling:

int getContentLength( )	Returns the size in bytes of the content associated with the resource. If the length is unavailable, –1 is returned.
long getContentLengthLong( )	Returns the size in bytes of the content associated with the resource. If the length is unavailable, –1 is returned.
String getContentType( )	Returns the type of content found in the resource. This is the value of the **content-type** header field. Returns **null** if the content type is not available.
long getDate( )	Returns the time and date of the response represented in terms of milliseconds since January 1, 1970 GMT.
long getExpiration( )	Returns the expiration time and date of the resource represented in terms of milliseconds since January 1, 1970 GMT. Zero is returned if the expiration date is unavailable.
String getHeaderField(int *idx*)	Returns the value of the header field at index *idx*. (Header field indexes begin at 0.) Returns **null** if the value of *idx* exceeds the number of fields.
String getHeaderField(String *fieldName*)	Returns the value of header field whose name is specified by *fieldName*. Returns **null** if the specified name is not found.
String getHeaderFieldKey(int *idx*)	Returns the header field key at index *idx*. (Header field indexes begin at 0.) Returns **null** if the value of *idx* exceeds the number of fields.
Map<String, List<String>> getHeaderFields( )	Returns a map that contains all of the header fields and values.
long getLastModified( )	Returns the time and date, represented in terms of milliseconds since January 1, 1970 GMT, of the last modification of the resource. Zero is returned if the last-modified date is unavailable.
InputStream getInputStream( ) throws IOException	Returns an **InputStream** that is linked to the resource. This stream can be used to obtain the content of the resource.

Notice that **URLConnection** defines several methods that handle header information. A header consists of pairs of keys and values represented as strings. By using **getHeaderField( )**, you can obtain the value associated with a header

key. By calling **getHeaderFields( )**, you can obtain a map that contains all of the headers. Several standard header fields are available directly through methods such as **getDate( )** and **getContentType( )**.

The following example creates a **URLConnection** using the **openConnection( )** method of a **URL** object and then uses it to examine the document's properties and content:

```
// Demonstrate URLConnection.
import java.net.*;
import java.io.*;
import java.util.Date;

class UCDemo
{
  public static void main(String args[]) throws Exception {
    int c;
    URL hp = new URL(http://www.internic.net");
    URLConnection hpCon = hp.openConnection();

    // get date
    long d = hpCon.getDate();
    if(d==0)
      System.out.println("No date information.");
    else
      System.out.println("Date: " + new Date(d));

    // get content type
    System.out.println("Content-Type: " + hpCon.getContentType());

    // get expiration date
    d = hpCon.getExpiration();
    if(d==0)
      System.out.println("No expiration information.");
    else
      System.out.println("Expires: " + new Date(d));

    // get last-modified date
    d = hpCon.getLastModified();
    if(d==0)
      System.out.println("No last-modified information.");
    else
      System.out.println("Last-Modified: " + new Date(d));

    // get content length
    long len = hpCon.getContentLengthLong();
    if(len == -1)
      System.out.println("Content length unavailable.");
    else
      System.out.println("Content-Length: " + len);

    if(len != 0) {
      System.out.println("=== Content ===");
      InputStream input = hpCon.getInputStream();
      while (((c = input.read()) != -1)) {
        System.out.print((char) c);
      }
      input.close();

    } else {
      System.out.println("No content available.");
    }
  }
}
```

The program establishes an HTTP connection to **www.internic.net** over port 80. It then displays several header values and retrieves the content. You might find it interesting to try this example, observing the results, and then for comparison purposes try different web sites of your own choosing.

HttpURLConnection

Java provides a subclass of **URLConnection** that provides support for HTTP connections. This class is called **HttpURLConnection.** You obtain an **HttpURLConnection** in the same way just shown, by calling **openConnection( )** on a **URL** object, but you must cast the result to **HttpURLConnection.** (Of course, you must make sure that you are actually opening an HTTP connection.) Once you have obtained a reference to an **HttpURLConnection** object, you can use any of the methods inherited from **URLConnection.** You can also use any of the several methods defined by **HttpURLConnection.** Here is a sampling:

static boolean getFollowRedirects( )	Returns **true** if redirects are automatically followed and **false** otherwise. This feature is on by default.
String getRequestMethod( )	Returns a string representing how URL requests are made. The default is GET. Other options, such as POST, are available.
int getResponseCode( ) throws IOException	Returns the HTTP response code. –1 is returned if no response code can be obtained. An **IOException** is thrown if the connection fails.
String getResponseMessage( ) throws IOException	Returns the response message associated with the response code. Returns **null** if no message is available. An **IOException** is thrown if the connection fails.
static void setFollowRedirects(boolean *how*)	If *how* is **true,** then redirects are automatically followed. If *how* is **false,** redirects are not automatically followed. By default, redirects are automatically followed.
void setRequestMethod(String *how*) throws ProtocolException	Sets the method by which HTTP requests are made to that specified by *how*. The default method is GET, but other options, such as POST, are available. If *how* is invalid, a **ProtocolException** is thrown.

The following program demonstrates **HttpURLConnection.** It first

establishes a connection to **www.google.com**. Then it displays the request method, the response code, and the response message. Finally, it displays the keys and values in the response header.

```
// Demonstrate HttpURLConnection.
import java.net.*;
import java.io.*;
import java.util.*;

class HttpURLDemo
{

public static void main(String args[]) throws Exception {
  URL hp = new URL(http://www.google.com");

  HttpURLConnection hpCon = (HttpURLConnection) hp.openConnection();

  // Display request method.
  System.out.println("Request method is " +
                     hpCon.getRequestMethod());

  // Display response code.
  System.out.println("Response code is " +
                     hpCon.getResponseCode());

  // Display response message.
  System.out.println("Response Message is " +
                     hpCon.getResponseMessage());

  // Get a list of the header fields and a set
  // of the header keys.
  Map<String, List<String>> hdrMap = hpCon.getHeaderFields();
  Set<String> hdrField = hdrMap.keySet();

  System.out.println("\nHere is the header:");

   // Display all header keys and values.
   for(String k : hdrField) {
     System.out.println("Key: " + k +
                        "  Value: " + hdrMap.get(k));
   }
  }
}
```

Here is a small portion of the output produced by the program. (Of course, the exact response returned by **www.google.com** will vary over time.)

```
Request method is GET
Response code is 200
Response Message is OK

Here is the header:
Key: Transfer-Encoding Value: [chunked]
Key: null Value: [HTTP/1.1 200 OK]
Key: Server Value: [gws]
```

Notice how the header keys and values are displayed. First, a map of the header keys and values is obtained by calling **getHeaderFields( )** (which is inherited from **URLConnection**). Next, a set of the header keys is retrieved by calling **keySet( )** on the map. Then, the key set is cycled through by using a for-each style **for** loop. The value associated with each key is obtained by calling **get( )** on the map.

The URI Class

The **URI** class encapsulates a *Uniform Resource Identifier (URI)*. URIs are similar to URLs. In fact, URLs constitute a subset of URIs. A URI represents a standard way to identify a resource. A URL also describes how to access the resource.

Cookies

The **java.net** package includes classes and interfaces that help manage cookies and can be used to create a stateful (as opposed to stateless) HTTP session. The classes are **CookieHandler, CookieManager,** and **HttpCookie.** The interfaces are **CookiePolicy** and **CookieStore.** The creation of a stateful HTTP session is beyond the scope of this book.

NOTE For information about using cookies with servlets, see Chapter 35.

TCP/IP Server Sockets

As mentioned earlier, Java has a different socket class that must be used for

creating server applications. The **ServerSocket** class is used to create servers that listen for either local or remote client programs to connect to them on published ports. **ServerSocket**s are quite different from normal **Socket**s. When you create a **ServerSocket,** it will register itself with the system as having an interest in client connections. The constructors for **ServerSocket** reflect the port number that you want to accept connections on and, optionally, how long you want the queue for said port to be. The queue length tells the system how many client connections it can leave pending before it should simply refuse connections. The default is 50. The constructors might throw an **IOException** under adverse conditions. Here are three of its constructors:

ServerSocket(int *port*) throws IOException	Creates server socket on the specified port with a queue length of 50.
ServerSocket(int *port*, int *maxQueue*) throws IOException	Creates a server socket on the specified port with a maximum queue length of *maxQueue*.
ServerSocket(int *port*, int *maxQueue*, InetAddress *localAddress*) throws IOException	Creates a server socket on the specified port with a maximum queue length of *maxQueue*. On a multihomed host, *localAddress* specifies the IP address to which this socket binds.

ServerSocket has a method called **accept( ),** which is a blocking call that will wait for a client to initiate communications and then return with a normal **Socket** that is then used for communication with the client.

Datagrams

TCP/IP-style networking is appropriate for most networking needs. It provides a serialized, predictable, reliable stream of packet data. This is not without its cost, however. TCP includes many complicated algorithms for dealing with congestion control on crowded networks, as well as pessimistic expectations about packet loss. This leads to a somewhat inefficient way to transport data. Datagrams provide an alternative.

Datagrams are bundles of information passed between machines. They are somewhat like a hard throw from a well-trained but blindfolded catcher to the third baseman. Once the datagram has been released to its intended target, there is no assurance that it will arrive or even that someone will be there to catch it. Likewise, when the datagram is received, there is no assurance that it hasn’t been damaged in transit or that whoever sent it is still there to receive a response.

Java implements datagrams on top of the UDP protocol by using two classes: the **DatagramPacket** object is the data container, while the **DatagramSocket** is the mechanism used to send or receive the **DatagramPacket**s. Each is examined here.

DatagramSocket

DatagramSocket defines four public constructors. They are shown here:

DatagramSocket( ) throws SocketException

DatagramSocket(int *port*) throws SocketException

DatagramSocket(int *port*, InetAddress *ipAddress*) throws SocketException

DatagramSocket(SocketAddress *address*) throws SocketException

The first creates a **DatagramSocket** bound to any unused port on the local computer. The second creates a **DatagramSocket** bound to the port specified by *port*. The third constructs a **DatagramSocket** bound to the specified port and **InetAddress**. The fourth constructs a **DatagramSocket** bound to the specified **SocketAddress**. **SocketAddress** is an abstract class that is implemented by the concrete class **InetSocketAddress**. **InetSocketAddress** encapsulates an IP address with a port number. All can throw a **SocketException** if an error occurs while creating the socket.

DatagramSocket defines many methods. Two of the most important are **send( )** and **receive( )**, which are shown here:

void send(DatagramPacket *packet*) throws IOException

void receive(DatagramPacket *packet*) throws IOException

The **send( )** method sends a packet to the port specified by *packet*. The **receive( )** method waits for a packet to be received and returns the result.

DatagramSocket also defines the **close( )**method, which closes the socket. **DatagramSocket** also implements **AutoCloseable**, which means that a **DatagramSocket** can be managed by a **try**-with-resources block.

Other methods give you access to various attributes associated with a **DatagramSocket**. Here is a sampling:

InetAddress getInetAddress( )	If the socket is connected, then the address is returned. Otherwise, **null** is returned.
int getLocalPort( )	Returns the number of the local port.
int getPort( )	Returns the number of the port connected to the socket. It returns –1 if the socket is not connected to a port.
boolean isBound( )	Returns **true** if the socket is bound to an address. Returns **false** otherwise.
boolean isConnected( )	Returns **true** if the socket is connected to a server. Returns **false** otherwise.
void setSoTimeout(int *millis*) throws SocketException	Sets the time-out period to the number of milliseconds passed in *millis*.

DatagramPacket

DatagramPacket defines several constructors. Four are shown here:

DatagramPacket(byte *data* [ ], int *size*)
DatagramPacket(byte *data* [ ], int *offset*, int *size*)
DatagramPacket(byte *data* [ ], int *size*, InetAddress *ipAddress*, int *port*)
DatagramPacket(byte *data* [ ], int *offset*, int *size*, InetAddress *ipAddress*, int *port*)

The first constructor specifies a buffer that will receive data and the size of a packet. It is used for receiving data over a **DatagramSocket.** The second form allows you to specify an offset into the buffer at which data will be stored. The third form specifies a target address and port, which are used by a **DatagramSocket** to determine where the data in the packet will be sent. The fourth form transmits packets beginning at the specified offset into the data. Think of the first two forms as building an "in box," and the second two forms as stuffing and addressing an envelope.

DatagramPacket defines several methods, including those shown here, that give access to the address and port number of a packet, as well as the raw data and its length:

InetAddress getAddress( )	Returns the address of the source (for datagrams being received) or destination (for datagrams being sent).
byte[ ] getData( )	Returns the byte array of data contained in the datagram. Mostly used to retrieve data from the datagram after it has been received.
int getLength( )	Returns the length of the valid data contained in the byte array that would be returned from the **getData( )** method. This may not equal the length of the whole byte array.
int getOffset( )	Returns the starting index of the data.
int getPort( )	Returns the port number.
void setAddress(InetAddress *ipAddress*)	Sets the address to which a packet will be sent. The address is specified by *ipAddress*.
void setData(byte[ ] *data*)	Sets the data to *data*, the offset to zero, and the length to number of bytes in *data*.
void setData(byte[ ] *data*, int *idx*, int *size*)	Sets the data to *data*, the offset to *idx*, and the length to *size*.
void setLength(int *size*)	Sets the length of the packet to *size*.
void setPort(int *port*)	Sets the port to *port*.

A Datagram Example

The following example implements a very simple networked communications client and server. Messages are typed into the window at the server and written across the network to the client side, where they are displayed.

```
// Demonstrate datagrams.
import java.net.*;

class WriteServer {
  public static int serverPort = 998;
  public static int clientPort = 999;
  public static int buffer_size = 1024;
  public static DatagramSocket ds;
  public static byte buffer[] = new byte[buffer_size];

  public static void TheServer() throws Exception {
    int pos=0;
    while (true) {
      int c = System.in.read();
      switch (c) {
        case -1:
          System.out.println("Server Quits.");
          ds.close();
          return;
        case '\r':
          break;
        case '\n':
          ds.send(new DatagramPacket(buffer,pos,
             InetAddress.getLocalHost(),clientPort));
          pos=0;
          break;
        default:
          buffer[pos++] = (byte) c;
      }
    }
  }

  public static void TheClient() throws Exception {
    while(true) {
      DatagramPacket p = new DatagramPacket(buffer, buffer.length);
      ds.receive(p);
      System.out.println(new String(p.getData(), 0, p.getLength()));
    }
  }

  public static void main(String args[]) throws Exception {
    if(args.length == 1) {
      ds = new DatagramSocket(serverPort);
      TheServer();
    } else {
      ds = new DatagramSocket(clientPort);
      TheClient();
    }
  }
}
```

This sample program is restricted by the **DatagramSocket** constructor to running between two ports on the local machine. To use the program, run

```
java WriteServer
```

in one window; this will be the client. Then run

```
java WriteServer 1
```

This will be the server. Anything that is typed in the server window will be sent to the client window after a newline is received.

NOTE The use of datagrams may not be allowed on your computer. (For example, a firewall may prevent their use.) If this is the case, the preceding example cannot be used. Also, the port numbers used in the program work on the author's system, but may have to be adjusted for your environment.

Introducing java.net.http

The preceding material introduced Java's traditional support for networking provided by **java.net**. This API is available in all versions of Java and is widely used. Thus, knowledge of Java's traditional approach to networking is important for all programmers. However, beginning with JDK 11, a new networking package called **java.net.http**, in the module **java.net.http**, has been added. It provides enhanced, updated networking support for HTTP clients. This new API is generally referred to as the *HTTP Client API*.

For many types of HTTP networking, the capabilities defined by the API in **java.net.http** can provide superior solutions. In addition to offering a streamlined, easy-to-use API, other advantages include support for asynchronous communication, HTTP/2, and flow control. In general, the HTTP Client API is designed as a superior alternative to the functionality provided by **HttpURLConnection**. It also supports the WebSocket protocol for bidirectional communication.

The following discussion explores several key features of the HTTP Client API. Be aware that it contains much more than described here. If you will be writing sophisticated network-based code, then it is a package that you will want to examine in detail. Our purpose here is to introduce some of the fundamentals associated with this important new module.

Three Key Elements

The focus of the following discussion is centered on three core HTTP Client API elements:

HttpClient	Encapsulates an HTTP client. It provides the means by which you send a request and obtain a response.
HttpRequest	Encapsulates a request.
HttpResponse	Encapsulates a response.

These work together to support the request/response features of HTTP. Here is the general procedure. First, create an instance of **HttpClient**. Then, construct an **HttpRequest** and send it by calling **send( )** on the **HttpClient**. The response is returned by **send( )**. From the response, you can obtain the headers and response body. Before working through an example, we will begin with an overview of these fundamental aspects of the API.

HttpClient

HttpClient encapsulates the HTTP request/response mechanism. It supports both synchronous and asynchronous communication. Here, we will be using only synchronous communication, but you might want to experiment with asynchronous communication on your own. Once you have an **HttpClient** object, you can use it to send requests and obtain responses. Thus, it is at the foundation of the HTTP Client API.

HttpClient is an abstract class, and instances are not created via a public constructor. Rather, you will use a factory method to build one. **HttpClient** supports *builders* with the **HttpClient.Builder** interface, which provides several methods that let you configure the **HttpClient**. To obtain an **HttpClient** builder, use the **newBuilder( )** static method. It returns a builder that lets you configure the **HttpClient** that it will create. Next, call **build( )** on the builder. It creates and returns the **HttpClient** instance. For example, this creates an **HttpClient** that uses the default settings:

```
HttpClient myHC = HttpClient.newBuilder().build();
```

HttpClient.Builder defines a number of methods that let you configure the builder. Here is one example. By default, redirects are not followed. You can change this by calling **followRedirects( )**, passing in the new redirect setting, which must be a value in the **HttpClient.Redirect** enumeration. It defines the

following values: **ALWAYS**, **NEVER**, and **NORMAL**. The first two are self explanatory. The **NORMAL** setting causes redirects to be followed unless a redirect is from an HTTPS site to an HTTP site. For example, this creates a builder in which the redirect policy is **NORMAL**. It then uses that builder to construct an **HttpClient**.

```
HttpClient.Builder myBuilder =
  HttpClient.newBuilder().followRedirects(HttpClient.Redirect.NORMAL
HttpClient.myHC = myBuilder.build();
```

Among others, builder configuration settings include authentication, proxy, HTTP version, and priority. Therefore, you can build an HTTP client to fit virtually any need.

In cases in which the default configuration is sufficient, you can obtain a default **HttpClient** directly by calling the **newHttpClient( )** method. It is shown here:

static HttpClient newHttpClient( )

An **HttpClient** with a default configuration is returned. For example, this creates a new default **HttpClient**:

```
HttpClient myHC = HttpClient.newHttpClient();
```

Because a default client is sufficient for the purposes of this book, this is the approach used by the examples that follow.

Once you have an **HttpClient** instance, you can send a synchronous request by calling its **send( )** method, shown here:

<T> HttpResponse <T> send(HttpRequest *req*,
HttpResponse.BodyHandler<T> *handler*)
throws IOException, InterruptedException

Here, *req* encapsulates the request and *handler* specifies how the response body is handled. As you will shortly see, often, you can use one of the predefined body handlers provided by the **HttpResponse.BodyHandlers** class. An **HttpResponse** object is returned. Thus, **send( )** provides the basic mechanism for HTTP communication.

HttpRequest

The HTTP Client API encapsulates requests in the **HttpRequest** abstract class.

To create an **HttpRequest** object, you will use a builder. To obtain a builder, call **newBuilder( )**. It has these two forms:

```
static HttpRequest.Builder newBuilder( )

static HttpRequest.Builder newBuilder(URI uri)
```

The first form creates a default builder. The second lets you specify the URI of the resource.

HttpRequest.Builder lets you specify various aspects of the request, such as what request method to use. (The default is GET.) You can also set header information, the URI, and the HTTP version, among others. Aside from the URI, often the default settings are sufficient. You can obtain a string representation of the request method by calling **method( )** on the **HttpRequest** object.

To actually construct a request, call **build( )** on the builder instance. It is shown here:

```
HttpRequest build( )
```

Once you have an **HttpRequest** instance, you can use it use it in a call to **HttpClient**'s **send( )** method, as shown in the previous section.

HttpResponse

The HTTP Client API encapsulates a response in an implementation of the **HttpResponse** interface. It is a generic interface declared like this:

```
HttpResponse<T>
```

Here, **T** specifies the type of body. Because the body type is generic, it enables the body to be handled in a variety of ways. This gives you a wide degree of flexibility in how your response code is written.

When a request is sent, an **HttpResponse** instance is returned that contains the response. **HttpResponse** defines several methods that give you access to the information in the response. Arguably, the most important is **body( )**, shown here:

```
T body( )
```

A reference to the body is returned. The specific type of reference is determined by the type of **T**, which is specified by the body handler specified by the **send( )** method.

You can obtain the status code associated with the response by calling **statusCode( )**, shown here:

int statusCode( )

The HTTP status code is returned. A value of 200 indicates success.

Another method in **HttpResponse** is **headers( )**, which obtains the response headers. It is shown here:

HttpHeaders headers( )

The headers associated with the response are encapsulated in an instance of the **HttpHeaders** class. It contains various methods that give you access to the headers. The one used by the example that follows is **map( )**, shown here:

Map<String, List<String>> map( )

It returns a map that contains all of the header fields and values.

One of the advantages of the HTTP Client API is that responses can be handled automatically and in a variety of ways. Responses are handled by implementations of the **HttpResponse.BodyHandler** interface. A number of predefined body handler factory methods are provided in the **HttpResponse.BodyHandlers** class. Here are three examples:

static HttpResponse.BodyHandler<Path> ofFile(Path *filename*)	Writes the body of the response to the file specified by *filename*. After the response is obtained, **HttpResponse.body( )** will return a **Path** to the file.
static HttpResponse.BodyHandler<InputStream> ofInputStream( )	Opens an **InputStream** to the response body. After the response is obtained, **HttpResponse.body( )** will return a reference to the **InputStream**.
static HttpResponse.BodyHandler<String> ofString( )	The body of the response is put in a string. After the response is obtained, **HttpResponse.body( )** returns the string.

Other predefined handlers obtain the response body as a byte array, a stream of lines, a download file, and a **Flow.Publisher**. A non-flow-controlled consumer is also supported. Before moving on, it is important to point out that the stream returned by **ofInputStream( )** should be read in its entirety. Doing so enables associated resources to be freed. If the entire body cannot be read for some reason, call **close( )** to close the stream, which may also close the HTTP connection. In general, it is best to simply read the entire stream.

A Simple HTTP Client Example

The following example puts into action the features of the HTTP Client API just described. It demonstrates the sending of a request, displaying the body of the response, and obtaining a list of the response headers. You should compare it to parallel sections of code in the preceding **UCDemo** and **HttpURLDemo** programs shown earlier. Notice that it uses **ofInputStream( )** to obtain an input stream linked to the response body.

```
// Demonstrate HttpClient.
import java.net.*;
import java.net.http.*;
import java.io.*;
import java.util.*;

class HttpClientDemo
{
  public static void main(String args[]) throws Exception {

    // Obtain a client that uses the default settings.
    HttpClient myHC = HttpClient.newHttpClient();

    // Create a request.
    HttpRequest myReq = HttpRequest.newBuilder(
                          new URI("http://www.google.com/")).build();

    // Send the request and get the response. Here, an InputStream is
    // used for the body.
    HttpResponse<InputStream> myResp = myHC.send(myReq,
                          HttpResponse.BodyHandlers.ofInputStream());

    // Display response code and response method.
    System.out.println("Response code is " + myResp.statusCode());
    System.out.println("Request method is " + myReq.method());

    // Get headers from the response.
    HttpHeaders hdrs = myResp.headers();

    // Get a map of the headers.
    Map<String, List<String>> hdrMap = hdrs.map();
    Set<String> hdrField = hdrMap.keySet();

    System.out.println("\nHere is the header:");

    // Display all header keys and values.
    for(String k : hdrField) {
      System.out.println("Key: " + k +
                         "  Value: " + hdrMap.get(k));
    }

    // Display the body.
    System.out.println("\nHere is the body: ");

    InputStream input = myResp.body();
    int c;
    // Read and display the entire body.
    while((c = input.read()) != -1) {
      System.out.print((char) c);
    }
  }
}
```

The program first creates an **HttpClient** and then uses that client to send a request to **www.google.com** (of course, you can substitute any website you like). The body handler uses an input stream by way of **ofInputStream( )**. Next, the response status code and the request method are displayed. Then, the header is displayed, followed by the body. Because **ofInputStream( )** was specified in the **send( )** method, the **body( )** method will return an **InputStream**. This stream is then used to read and display the body.

The preceding program used an input stream to handle the body for comparison purposes with the **UCDemo** program shown earlier, which uses a parallel approach. However, other options are available. For example, you can use **ofString( )** to handle the body as a string. With this approach, when the response is obtained, the body will be in a **String** instance. To try this, first substitute the line that calls **send( )** with the following:

```
HttpResponse<InputStream> myResp = myHC.send(myReq,
                        HttpResponse.BodyHandlers.ofString());
```

Next, replace the code that uses an input stream to read and display the body with the following line:

```
System.out.println(myResp.body());
```

Because the body of the response is already stored in a string, it can be output directly. You might want to experiment with other body handlers. Of particular interest is **ofLines( )**, which lets you access the body as a stream of lines. One of the benefits of the HTPP Client API is that there are built-in body handlers for a variety of situations.

Things to Explore in java.net.http

The preceding introduction described a number of key features in the HTTP Client API in **java.net.http**, but there are several more that you will want to explore. One of the most important is the **WebSocket** class, which supports bidirectional communication. Another is the asynchronous capability supported by the API. In general, if network programming is in your future, you will want to thoroughly explore **java.net.http**. It is an important addition to Java's networking APIs.

CHAPTER

24

Event Handling

This chapter examines an important aspect of Java: the event. Event handling is fundamental to Java programming because it is integral to the creation of many kinds of applications. For example, any program that uses a graphical user interface, such as a Java application written for Windows, is event driven. Thus, you cannot write these types of programs without a solid command of event handling. Events are supported by a number of packages, including **java.util**, **java.awt**, and **java.awt.event**. Beginning with JDK 9, **java.awt** and **java.awt.event** are part of the **java.desktop** module, and **java.util** is part of the **java.base** module.

Many events to which your program will respond are generated when the user interacts with a GUI-based program. These are the types of events examined in this chapter. They are passed to your program in a variety of ways, with the specific method dependent upon the actual event. There are several types of events, including those generated by the mouse, the keyboard, and various GUI controls, such as a push button, scroll bar, or check box.

This chapter begins with an overview of Java's event handling mechanism. It then examines a number of event classes and interfaces used by the Abstract Window Toolkit (AWT). The AWT was Java's first GUI framework and it offers a simple way to present the basics of event handling. Next, the chapter develops several examples that demonstrate the fundamentals of event processing. This chapter also introduces key concepts related to GUI programming, and explains how to use adapter classes, inner classes, and anonymous inner classes to streamline event handling code. The examples provided in the remainder of this book make frequent use of these techniques.

NOTE This chapter focuses on events related to GUI-based programs. However, events are also occasionally used for purposes not directly related to GUI-based programs. In all cases, the same basic event handling techniques apply.

Two Event Handling Mechanisms

Before beginning our discussion of event handling, an important historical point must be made: The way in which events are handled changed significantly

between the original version of Java (1.0) and all subsequent versions of Java, beginning with version 1.1. Although the 1.0 method of event handling is still supported, it is not recommended for new programs. Also, many of the methods that support the old 1.0 event model have been deprecated. The modern approach is the way that events should be handled by all new programs and thus is the method employed by programs in this book.

The Delegation Event Model

The modern approach to handling events is based on the *delegation event model,* which defines standard and consistent mechanisms to generate and process events. Its concept is quite simple: a *source* generates an event and sends it to one or more *listeners.* In this scheme, the listener simply waits until it receives an event. Once an event is received, the listener processes the event and then returns. The advantage of this design is that the application logic that processes events is cleanly separated from the user interface logic that generates those events. A user interface element is able to "delegate" the processing of an event to a separate piece of code.

In the delegation event model, listeners must register with a source in order to receive an event notification. This provides an important benefit: notifications are sent only to listeners that want to receive them. This is a more efficient way to handle events than the design used by the original Java 1.0 approach. Previously, an event was propagated up the containment hierarchy until it was handled by a component. This required components to receive events that they did not process, and it wasted valuable time. The delegation event model eliminates this overhead.

The following sections define events and describe the roles of sources and listeners.

Events

In the delegation model, an *event* is an object that describes a state change in a source. Among other causes, an event can be generated as a consequence of a person interacting with the elements in a graphical user interface. Some of the activities that cause events to be generated are pressing a button, entering a character via the keyboard, selecting an item in a list, and clicking the mouse. Many other user operations could also be cited as examples.

Events may also occur that are not directly caused by interactions with a user

interface. For example, an event may be generated when a timer expires, a counter exceeds a value, a software or hardware failure occurs, or an operation is completed. You are free to define events that are appropriate for your application.

Event Sources

A *source* is an object that generates an event. This occurs when the internal state of that object changes in some way. Sources may generate more than one type of event.

A source must register listeners in order for the listeners to receive notifications about a specific type of event. Each type of event has its own registration method. Here is the general form:

public void add*Type*Listener (*Type*Listener *el*)

Here, *Type* is the name of the event, and *el* is a reference to the event listener. For example, the method that registers a keyboard event listener is called **addKeyListener( )**. The method that registers a mouse motion listener is called **addMouseMotionListener( )**. When an event occurs, all registered listeners are notified and receive a copy of the event object. This is known as *multicasting* the event. In all cases, notifications are sent only to listeners that register to receive them.

Some sources may allow only one listener to register. The general form of such a method is this:

public void add*Type*Listener(*Type*Listener *el*)
throws java.util.TooManyListenersException

Here, *Type* is the name of the event, and *el* is a reference to the event listener. When such an event occurs, the registered listener is notified. This is known as *unicasting* the event.

A source must also provide a method that allows a listener to unregister an interest in a specific type of event. The general form of such a method is this:

public void remove*Type*Listener(*Type*Listener *el*)

Here, *Type* is the name of the event, and *el* is a reference to the event listener. For example, to remove a keyboard listener, you would call **removeKeyListener( )**.

The methods that add or remove listeners are provided by the source that generates events. For example, **Component**, which is a top-level class defined by the AWT, provides methods to add and remove keyboard and mouse event listeners.

Event Listeners

A *listener* is an object that is notified when an event occurs. It has two major requirements. First, it must have been registered with one or more sources to receive notifications about specific types of events. Second, it must implement methods to receive and process these notifications. In other words, the listener must supply the event handlers.

The methods that receive and process events are defined in a set of interfaces, such as those found in **java.awt.event**. For example, the **MouseMotionListener** interface defines two methods to receive notifications when the mouse is dragged or moved. Any object may handle one or both of these events if it provides an implementation of this interface. Other listener interfaces are discussed later in this and other chapters.

Here is one more key point about events: An event handler must return quickly. For the most part, a GUI program should not enter a "mode" of operation in which it maintains control for an extended period. Instead, it must perform specific actions in response to events and then return control to the run-time system. Failure to do this can cause your program to appear sluggish or even non-responsive. If your program needs to perform a repetitive task, such as scrolling a banner, it must do so by starting a separate thread. In short, when your program receives an event, it must process it immediately, and then return.

Event Classes

The classes that represent events are at the core of Java's event handling mechanism. Thus, a discussion of event handling must begin with the event classes. It is important to understand, however, that Java defines several types of events and that not all event classes can be discussed in this chapter. Arguably, the most widely used events at the time of this writing are those defined by the AWT and those defined by Swing. This chapter focuses on the AWT events. (Most of these events also apply to Swing.) Several Swing-specific events are described in Chapter 31, when Swing is covered.

At the root of the Java event class hierarchy is **EventObject**, which is in

java.util. It is the superclass for all events. Its one constructor is shown here:

EventObject(Object *src*)

Here, *src* is the object that generates this event.

EventObject defines two methods: **getSource( )** and **toString( )**. The **getSource( )** method returns the source of the event. Its general form is shown here:

Object getSource( )

As expected, **toString( )** returns the string equivalent of the event.

The class **AWTEvent**, defined within the **java.awt** package, is a subclass of **EventObject**. It is the superclass (either directly or indirectly) of all AWT-based events used by the delegation event model. Its **getID( )** method can be used to determine the type of the event. The signature of this method is shown here:

int getID( )

Typically, you won't use the features defined by **AWTEvent** directly. Rather, you will use its subclasses. At this point, it is important to know only that all of the other classes discussed in this section are subclasses of **AWTEvent**.

To summarize:

- **EventObject** is a superclass of all events.
- **AWTEvent** is a superclass of all AWT events that are handled by the delegation event model.

The package **java.awt.event** defines many types of events that are generated by various user interface elements. Table 24-1 shows several commonly used event classes and provides a brief description of when they are generated. Commonly used constructors and methods in each class are described in the following sections.

Event Class	Description
ActionEvent	Generated when a button is pressed, a list item is double-clicked, or a menu item is selected.
AdjustmentEvent	Generated when a scroll bar is manipulated.
ComponentEvent	Generated when a component is hidden, moved, resized, or becomes visible.
ContainerEvent	Generated when a component is added to or removed from a container.
FocusEvent	Generated when a component gains or loses keyboard focus.
InputEvent	Abstract superclass for all component input event classes.
ItemEvent	Generated when a check box or list item is clicked; also occurs when a choice selection is made or a checkable menu item is selected or deselected.
KeyEvent	Generated when input is received from the keyboard.
MouseEvent	Generated when the mouse is dragged, moved, clicked, pressed, or released; also generated when the mouse enters or exits a component.
MouseWheelEvent	Generated when the mouse wheel is moved.
TextEvent	Generated when the value of a text area or text field is changed.
WindowEvent	Generated when a window is activated, closed, deactivated, deiconified, iconified, opened, or quit.

Table 24-1 Commonly Used Event Classes in **java.awt.event**

The ActionEvent Class

An **ActionEvent** is generated when a button is pressed, a list item is double-clicked, or a menu item is selected. The **ActionEvent** class defines four integer constants that can be used to identify any modifiers associated with an action event: **ALT_MASK**, **CTRL_MASK**, **META_MASK**, and **SHIFT_MASK**. In addition, there is an integer constant, **ACTION_PERFORMED**, which can be used to identify action events.

ActionEvent has these three constructors:

ActionEvent(Object *src*, int *type*, String *cmd*)
ActionEvent(Object *src*, int *type*, String *cmd*, int *modifiers*)
ActionEvent(Object *src*, int *type*, String *cmd*, long *when*, int *modifiers*)

Here, *src* is a reference to the object that generated this event. The type of the event is specified by *type*, and its command string is *cmd*. The argument *modifiers* indicates which modifier keys (ALT, CTRL, META, and/or SHIFT) were pressed when the event was generated. The *when* parameter specifies when the event occurred.

You can obtain the command name for the invoking **ActionEvent** object by using the **getActionCommand( )** method, shown here:

String getActionCommand( )

For example, when a button is pressed, an action event is generated that has a command name equal to the label on that button.

The **getModifiers( )** method returns a value that indicates which modifier keys (ALT, CTRL, META, and/or SHIFT) were pressed when the event was generated. Its form is shown here:

int getModifiers( )

The method **getWhen( )** returns the time at which the event took place. This is called the event's *timestamp*. The **getWhen( )** method is shown here:

long getWhen( )

The AdjustmentEvent Class

An **AdjustmentEvent** is generated by a scroll bar. There are five types of adjustment events. The **AdjustmentEvent** class defines integer constants that can be used to identify them. The constants and their meanings are shown here:

BLOCK_DECREMENT	The user clicked inside the scroll bar to decrease its value.
BLOCK_INCREMENT	The user clicked inside the scroll bar to increase its value.
TRACK	The slider was dragged.
UNIT_DECREMENT	The button at the end of the scroll bar was clicked to decrease its value.
UNIT_INCREMENT	The button at the end of the scroll bar was clicked to increase its value.

In addition, there is an integer constant, **ADJUSTMENT_VALUE_CHANGED**, that indicates that a change has occurred.

Here is one **AdjustmentEvent** constructor:

AdjustmentEvent(Adjustable *src*, int *id*, int *type*, int *val*)

Here, *src* is a reference to the object that generated this event. The *id* specifies