**NOTE** It is necessary to state that the term *stream* as used here differs from the use of *stream* when the I/O classes were described earlier in this book. Although an I/O stream can act conceptually much like one of the streams defined by **java.util.stream**, they are not the same. Thus, throughout this chapter, when the term *stream* is used, it refers to objects based on one of the stream types described here.

## Stream Interfaces

The stream API defines several stream interfaces, which are packaged in **java.util.stream**. At the foundation is **BaseStream**, which defines the basic functionality available in all streams. **BaseStream** is a generic interface declared like this:

```
interface BaseStream<T, S extends BaseStream<T, S>>
```

Here, **T** specifies the type of the elements in the stream, and **S** specifies the type of stream that extends **BaseStream**. **BaseStream** extends the **AutoCloseable** interface; thus, a stream can be managed in a **try**-with-resources statement. In general, however, only those streams whose data source requires closing (such as those connected to a file) will need to be closed. In most cases, such as those in which the data source is a collection, there is no need to close the stream. The methods declared by **BaseStream** are shown in Table 29-1.

| Method | Description |
|---|---|
| void close( ) | Closes the invoking stream, calling any registered close handlers. (As explained in the text, few streams need to be closed.) |
| boolean isParallel( ) | Returns **true** if the invoking stream is parallel. Returns **false** if the stream is sequential. |
| Iterator<T> iterator( ) | Obtains an iterator to the stream and returns a reference to it. (Terminal operation.) |
| S onClose(Runnable *handler*) | Returns a new stream with the close handler specified by *handler*. This handler will be called when the stream is closed. (Intermediate operation.) |
| S parallel( ) | Returns a parallel stream based on the invoking stream. If the invoking stream is already parallel, then that stream is returned. (Intermediate operation.) |
| S sequential( ) | Returns a sequential stream based on the invoking stream. If the invoking stream is already sequential, then that stream is returned. (Intermediate operation.) |
| Spliterator<T> spliterator( ) | Obtains a spliterator to the stream and returns a reference to it. (Terminal operation.) |
| S unordered( ) | Returns an unordered stream based on the invoking stream. If the invoking stream is already unordered, then that stream is returned. (Intermediate operation.) |

**Table 29-1** The Methods Declared by **BaseStream**

From **BaseStream** are derived several types of stream interfaces. The most general of these is **Stream**. It is declared as shown here:

interface Stream<T>

Here, **T** specifies the type of the elements in the stream. Because it is generic, **Stream** is used for all reference types. In addition to the methods that it inherits from **BaseStream**, the **Stream** interface adds several of its own, a sampling of which is shown in Table 29-2.

| Method | Description |
|---|---|
| <R, A> R collect(Collector<? super T, A, R> *collectorFunc*) | Collects elements into a container, which is changeable, and returns the container. This is called a mutable reduction operation. Here, **R** specifies the type of the resulting container and **T** specifies the element type of the invoking stream. **A** specifies the internal accumulated type. The *collectorFunc* specifies how the collection process works. (Terminal operation.) |
| long count( ) | Counts the number of elements in the stream and returns the result. (Terminal operation.) |
| Stream<T> filter(Predicate<? super T> *pred*) | Produces a stream that contains those elements from the invoking stream that satisfy the predicate specified by *pred*. (Intermediate operation.) |
| void forEach(Consumer<? super T> *action*) | For each element in the invoking stream, the code specified by *action* is executed. (Terminal operation.) |
| <R> Stream<R> map(Function<? super T, ? extends R> *mapFunc*) | Applies *mapFunc* to the elements from the invoking stream, yielding a new stream that contains those elements. (Intermediate operation.) |
| DoubleStream mapToDouble( ToDoubleFunction<? super T> *mapFunc*) | Applies *mapFunc* to the elements from the invoking stream, yielding a new **DoubleStream** that contains those elements. (Intermediate operation.) |
| IntStream mapToInt( ToIntFunction<? super T> *mapFunc*) | Applies *mapFunc* to the elements from the invoking stream, yielding a new **IntStream** that contains those elements. (Intermediate operation.) |
| LongStream mapToLong( ToLongFunction<? super T> *mapFunc*) | Applies *mapFunc* to the elements from the invoking stream, yielding a new **LongStream** that contains those elements. (Intermediate operation.) |
| Optional<T> max( Comparator<? super T> *comp*) | Using the ordering specified by *comp*, finds and returns the maximum element in the invoking stream. (Terminal operation.) |

**Table 29-2** A Sampling of Methods Declared by **Stream**

| Method | Description |
|---|---|
| Optional<T> min(Comparator<? super T> *comp*) | Using the ordering specified by *comp*, finds and returns the minimum element in the invoking stream. (Terminal operation.) |
| T reduce(T *identityVal*,<br>BinaryOperator<T> *accumulator*) | Returns a result based on the elements in the invoking stream. This is called a reduction operation. (Terminal operation.) |
| Stream<T> sorted( ) | Produces a new stream that contains the elements of the invoking stream sorted in natural order. (Intermediate operation.) |
| Object[] toArray( ) | Creates an array from the elements in the invoking stream. (Terminal operation.) |

**Table 29-2** A Sampling of Methods Declared by **Stream** *(continued)*

In both tables, notice that many of the methods are notated as being either *terminal* or *intermediate*. The difference between the two is very important. A *terminal* operation consumes the stream. It is used to produce a result, such as finding the minimum value in the stream, or to execute some action, as is the case with the **forEach( )** method. Once a stream has been consumed, it cannot be reused. *Intermediate* operations produce another stream. Thus, intermediate operations can be used to create a *pipeline* that performs a sequence of actions. One other point: intermediate operations do not take place immediately. Instead, the specified action is performed when a terminal operation is executed on the new stream created by an intermediate operation. This mechanism is referred to as *lazy behavior*, and the intermediate operations are referred to as *lazy*. The use of lazy behavior enables the stream API to perform more efficiently.

Another key aspect of streams is that some intermediate operations are *stateless* and some are *stateful*. In a stateless operation, each element is processed independently of the others. In a stateful operation, the processing of an element may depend on aspects of the other elements. For example, sorting is a stateful operation because an element's order depends on the values of the other elements. Thus, the **sorted( )** method is stateful. However, filtering elements based on a stateless predicate is stateless because each element is handled individually. Thus, **filter( )** can (and should be) stateless. The difference between stateless and stateful operations is especially important when parallel processing of a stream is desired because a stateful operation may require more than one pass to complete.

Because **Stream** operates on object references, it can't operate directly on primitive types. To handle primitive type streams, the stream API defines the following interfaces:

DoubleStream

IntStream

LongStream

These streams all extend **BaseStream** and have capabilities similar to **Stream** except that they operate on primitive types rather than reference types. They also provide some convenience methods, such as **boxed( )**, that facilitate their use. Because streams of objects are the most common, **Stream** is the primary focus of this chapter, but the primitive type streams can be used in much the same way.

## How to Obtain a Stream

You can obtain a stream in a number of ways. Perhaps the most common is when a stream is obtained for a collection. Beginning with JDK 8, the **Collection** interface has been expanded to include two methods that obtain a stream from a collection. The first is **stream( )**, shown here:

default Stream<E> stream( )

Its default implementation returns a sequential stream. The second method is **parallelStream( )**, shown next:

default Stream<E> parallelStream( )

Its default implementation returns a parallel stream, if possible. (If a parallel stream can not be obtained, a sequential stream may be returned instead.) Parallel streams support parallel execution of stream operations. Because **Collection** is implemented by every collection, these methods can be used to obtain a stream from any collection class, such as **ArrayList** or **HashSet**.

A stream can also be obtained from an array by use of the static **stream( )** method, which was added to the **Arrays** class by JDK 8. One of its forms is shown here:

static <T> Stream<T> stream(T[ ] *array*)

This method returns a sequential stream to the elements in *array*. For example, given an array called **addresses** of type **Address**, the following obtains a stream to it:

```
Stream<Address> addrStrm = Arrays.stream(addresses);
```

Several overloads of the **stream( )** method are also defined, such as those that handle arrays of the primitive types. They return a stream of type **IntStream**, **DoubleStream**, or **LongStream**.

Streams can be obtained in a variety of other ways. For example, many stream operations return a new stream, and a stream to an I/O source can be obtained by calling **lines( )** on a **BufferedReader**. However a stream is obtained, it can be used in the same way as any other stream.

## A Simple Stream Example

Before going any further, let's work through an example that uses streams. The following program creates an **ArrayList** called **myList** that holds a collection of integers (which are automatically boxed into the **Integer** reference type). Next, it obtains a stream that uses **myList** as a source. It then demonstrates various stream operations.

```
// Demonstrate several stream operations.

import java.util.*;
import java.util.stream.*;

class StreamDemo {

  public static void main(String[] args) {
```

```
    // Create a list of Integer values.
    ArrayList<Integer> myList = new ArrayList<>( );
    myList.add(7);
    myList.add(18);
    myList.add(10);
    myList.add(24);
    myList.add(17);
    myList.add(5);

    System.out.println("Original list: " + myList);

    // Obtain a Stream to the array list.
    Stream<Integer> myStream = myList.stream();

    // Obtain the minimum and maximum value by use of min(),
    // max(), isPresent(), and get().
    Optional<Integer> minVal = myStream.min(Integer::compare);
    if(minVal.isPresent()) System.out.println("Minimum value: " +
                                              minVal.get());

    // Must obtain a new stream because previous call to min()
    // is a terminal operation that consumed the stream.
    myStream = myList.stream();
    Optional<Integer> maxVal = myStream.max(Integer::compare);
    if(maxVal.isPresent()) System.out.println("Maximum value: " +
                                              maxVal.get());

    // Sort the stream by use of sorted().
    Stream<Integer> sortedStream = myList.stream().sorted();

    // Display the sorted stream by use of forEach().
    System.out.print("Sorted stream: ");
    sortedStream.forEach((n) -> System.out.print(n + " "));
    System.out.println();

    // Display only the odd values by use of filter().
    Stream<Integer> oddVals =
           myList.stream().sorted().filter((n) -> (n % 2) == 1);
    System.out.print("Odd values: ");
    oddVals.forEach((n) -> System.out.print(n + " "));
    System.out.println();

    // Display only the odd values that are greater than 5. Notice that
    // two filter operations are pipelined.
    oddVals = myList.stream().filter( (n) -> (n % 2) == 1)
                            .filter((n) -> n > 5);
    System.out.print("Odd values greater than 5: ");
    oddVals.forEach((n) -> System.out.print(n + " ") );
    System.out.println();
  }
}
```

The output is shown here:

```
Original list: [7, 18, 10, 24, 17, 5]
Minimum value: 5
Maximum value: 24
Sorted stream: 5 7 10 17 18 24
Odd values: 5 7 17
Odd values greater than 5: 7 17
```

Let's look closely at each stream operation. After creating an **ArrayList**, the program obtains a stream for the list by calling **stream( )**, as shown here:

```
Stream<Integer> myStream = myList.stream();
```

As explained, the **Collection** interface now defines the **stream( )** method, which obtains a stream from the invoking collection. Because **Collection** is implemented by every collection class, **stream( )** can be used to obtain stream for any type of collection, including the **ArrayList** used here. In this case, a reference to the stream is assigned to **myStream**.

Next, the program obtains the minimum value in the stream (which is, of course, also the minimum value in the data source) and displays it, as shown here:

```
Optional<Integer> minVal = myStream.min(Integer::compare);
if(minVal.isPresent()) System.out.println("Minimum value: " +
                                          minVal.get());
```

Recall from Table 29-2 that **min( )** is declared like this:

Optional<T> min(Comparator<? super T> *comp*)

First, notice that the type of **min( )**'s parameter is a **Comparator**. This comparator is used to compare two elements in the stream. In the example, **min( )** is passed a method reference to **Integer**'s **compare( )** method, which is used to implement a **Comparator** capable of comparing two **Integer**s. Next, notice that the return type of **min( )** is **Optional**. The **Optional** class is described in Chapter 19, but briefly, here is how it works. **Optional** is a generic class packaged in **java.util** and declared like this:

class Optional<T>

Here, **T** specifies the element type. An **Optional** instance can either contain a value of type **T** or be empty. You can use **isPresent( )** to determine if a value is present. Assuming that a value is available, it can be obtained by calling **get( )**. In this example, the object returned will hold the minimum value of the stream as an **Integer** object.

One other point about the preceding line: **min( )** is a terminal operation that consumes the stream. Thus, **myStream** cannot be used again after **min( )** executes.

The next lines obtain and display the maximum value in the stream:

```
myStream = myList.stream();
Optional<Integer> maxVal = myStream.max(Integer::compare);
if(maxVal.isPresent()) System.out.println("Maximum value: " +
                                          maxVal.get());
```

First, **myStream** is once again assigned the stream returned by **myList.stream( )**. As just explained, this is necessary because the previous call to **min( )** consumed the previous stream. Thus, a new one is needed. Next, the **max( )** method is called to obtain the maximum value. Like **min( )**, **max( )** returns an **Optional** object. Its value is obtained by calling **get( )**.

The program then obtains a sorted stream through the use of this line:

```
Stream<Integer> sortedStream = myList.stream().sorted();
```

Here, the **sorted( )** method is called on the stream returned by **myList.stream( )**. Because **sorted( )** is an intermediate operation, its result is a new stream, and this is the stream assigned to **sortedStream**. The contents of the sorted stream are displayed by use of **forEach( )**:

```
sortedStream.forEach((n) -> System.out.print(n + " "));
```

Here, the **forEach( )** method executes an operation on each element in the stream. In this case, it simply calls **System.out.print( )** for each element in **sortedStream**. This is accomplished by use of a lambda expression. The **forEach( )** method has this general form:

void forEach(Consumer<? super T> *action*)

**Consumer** is a generic functional interface declared in **java.util.function**. Its abstract method is **accept( )**, shown here:

void accept(T *objRef*)

The lambda expression in the call to **forEach( )** provides the implementation of **accept( )**. The **forEach( )** method is a terminal operation. Thus, after it completes, the stream has been consumed.

Next, a sorted stream is filtered by **filter( )** so that it contains only odd values:

```
Stream<Integer> oddVals =
       myList.stream().sorted().filter((n) -> (n % 2) == 1);
```

The **filter( )** method filters a stream based on a predicate. It returns a new stream that contains only those elements that satisfy the predicate. It is shown here:

Stream<T> filter(Predicate<? super T> *pred*)

**Predicate** is a generic functional interface defined in **java.util.function**. Its abstract method is **test( )**, which is shown here:

boolean test(T *objRef*)

It returns **true** if the object referred to by *objRef* satisfies the predicate, and **false** otherwise. The lambda expression passed to **filter( )** implements this method. Because **filter( )** is an intermediate operation, it returns a new stream that contains filtered values, which, in this case, are the odd numbers. These elements are then displayed via **forEach( )** as before.

Because **filter( )**, or any other intermediate operation, returns a new stream, it is possible to filter a filtered stream a second time. This is demonstrated by the following line, which produces a stream that contains only those odd values greater than 5:

```
oddVals = myList.stream().filter((n) -> (n % 2) == 1)
                         .filter((n) -> n > 5);
```

Notice that lambda expressions are passed to both filters.

## Reduction Operations

Consider the **min( )** and **max( )** methods in the preceding example program. Both are terminal operations that return a result based on the elements in the stream. In the language of the stream API, they represent *reduction operations* because each reduces a stream to a single value—in this case, the minimum and maximum. The stream API refers to these as *special case* reductions because they perform a specific function. In addition to **min( )** and **max( )**, other special case reductions are also available, such as **count( )**, which counts the number of elements in a stream. However, the stream API generalizes this concept by providing the **reduce( )** method. By using **reduce( )**, you can return a value from a stream based on any arbitrary criteria. By definition, all reduction operations are terminal operations.

**Stream** defines three versions of **reduce( )**. The two we will use first are shown here:

Optional<T> reduce(BinaryOperator<T> *accumulator*)

T reduce(T *identityVal*, BinaryOperator<T> *accumulator*)

The first form returns an object of type **Optional**, which contains the result. The second form returns an object of type **T** (which is the element type of the stream). In both forms, *accumulator* is a function that operates on two values and produces a result. In the second form, *identityVal* is a value such that an accumulator operation involving *identityVal* and any element of the stream yields that element, unchanged. For example, if the operation is addition, then the identity value will be 0 because 0 + x is x. For multiplication, the value will be 1, because 1 * x is x.

**BinaryOperator** is a functional interface declared in **java.util.function** that extends the **BiFunction** functional interface. **BiFunction** defines this abstract method:

R apply(T *val*, U *val2*)

Here, **R** specifies the result type, **T** is the type of the first operand, and **U** is the type of second operand. Thus, **apply( )** applies a function to its two operands (*val* and *val2*) and returns the result. When **BinaryOperator** extends **BiFunction**, it specifies the same type for all the type parameters. Thus, as it relates to **BinaryOperator**, **apply( )** looks like this:

T apply(T *val*, T *val2*)

Furthermore, as it relates to **reduce( )**, *val* will contain the previous result and *val2* will contain the next element. In its first invocation, *val* will contain either the identity value or the first element, depending on which version of **reduce( )** is used.

It is important to understand that the accumulator operation must satisfy three constraints. It must be

- Stateless
- Non-interfering
- Associative

As explained earlier, *stateless* means that the operation does not rely on any state information. Thus, each element is processed independently. *Non-interfering* means that the data source is not modified by the operation. Finally, the operation must be *associative*. Here, the term *associative* is used in its normal, arithmetic sense, which means that, given an associative operator used in a sequence of operations, it does not matter which pair of operands are processed first. For example,

(10 * 2) * 7

yields the same result as

10 * (2 * 7)

Associativity is of particular importance to the use of reduction operations on parallel streams, discussed in the next section.

The following program demonstrates the versions of **reduce( )** just described:

```
// Demonstrate the reduce() method.

import java.util.*;
import java.util.stream.*;

class StreamDemo2 {

  public static void main(String[] args) {

    // Create a list of Integer values.
    ArrayList<Integer> myList = new ArrayList<>( );

    myList.add(7);
    myList.add(18);
    myList.add(10);
    myList.add(24);
    myList.add(17);
    myList.add(5);

    // Two ways to obtain the integer product of the elements
    // in myList by use of reduce().
    Optional<Integer> productObj = myList.stream().reduce((a,b) -> a*b);
    if(productObj.isPresent())
      System.out.println("Product as Optional: " + productObj.get());

    int product = myList.stream().reduce(1, (a,b) -> a*b);
    System.out.println("Product as int: " + product);
  }
}
```

As the output here shows, both uses of **reduce( )** produce the same result:

```
Product as Optional: 2570400
Product as int: 2570400
```

In the program, the first version of **reduce( )** uses the lambda expression to produce a product of two values. In this case, because the stream contains **Integer** values, the **Integer** objects are automatically unboxed for the multiplication and reboxed to return the result. The two values represent the current value of the running result and the next element in the stream. The final result is returned in an object of type **Optional**. The value is obtained by calling **get( )** on the returned object.

In the second version, the identity value is explicitly specified, which for multiplication is 1. Notice that the result is returned as an object of the element type, which is **Integer** in this case.

Although simple reduction operations such as multiplication are useful for examples, reductions are not limited in this regard. For example, assuming the preceding program, the following obtains the product of only the even values:

```
int evenProduct = myList.stream().reduce(1, (a,b) -> {
                    if(b%2 == 0) return a*b; else return a;
                  });
```

Pay special attention to the lambda expression. If **b** is even, then **a** * **b** is returned. Otherwise, **a** is returned. This works because **a** holds the current result and **b** holds the next element, as explained earlier.

## Using Parallel Streams

Before exploring any more of the stream API, it will be helpful to discuss parallel streams. As has been pointed out previously in this book, the parallel execution of code via multicore processors can result in a substantial increase in performance. Because of this, parallel programming has become an important part of the modern programmer's job. However, parallel programming can be complex and error-prone. One of the benefits that the stream library offers is the ability to easily—and reliably—parallel process certain operations.

Parallel processing of a stream is quite simple to request: just use a parallel stream. As mentioned earlier, one way to obtain a parallel stream is to use the **parallelStream( )** method defined by **Collection**. Another way to obtain a parallel stream is to call the **parallel( )** method on a sequential stream. The **parallel( )** method is defined by **BaseStream**, as shown here:

S parallel()

It returns a parallel stream based on the sequential stream that invokes it. (If it is called on a stream that is already parallel, then the invoking stream is returned.) Understand, of course, that even with a parallel stream, parallelism will be achieved only if the environment supports it.

Once a parallel stream has been obtained, operations on the stream can occur in parallel, assuming that parallelism is supported by the environment. For example, the first **reduce( )**

operation in the preceding program can be parallelized by substituting **parallelStream( )** for the call to **stream( )**:

```
Optional<Integer> productObj = myList.parallelStream().reduce((a,b) -> a*b);
```

The results will be the same, but the multiplications can occur in different threads.

As a general rule, any operation applied to a parallel stream must be stateless. It should also be non-interfering and associative. This ensures that the results obtained by executing operations on a parallel stream are the same as those obtained from executing the same operations on a sequential stream.

When using parallel streams, you might find the following version of **reduce( )** especially helpful. It gives you a way to specify how partial results are combined:

```
<U> U reduce(U identityVal, BiFunction<U, ? super T, U> accumulator
             BinaryOperator<U> combiner)
```

In this version, *combiner* defines the function that combines two values that have been produced by the *accumulator* function. Assuming the preceding program, the following statement computes the product of the elements in **myList** by use of a parallel stream:

```
int parallelProduct = myList.parallelStream().reduce(1, (a,b) -> a*b,
                                                        (a,b) -> a*b);
```

As you can see, in this example, both the accumulator and combiner perform the same function. However, there are cases in which the actions of the accumulator must differ from those of the combiner. For example, consider the following program. Here, **myList** contains a list of **double** values. It then uses the combiner version of **reduce( )** to compute the product of the *square roots* of each element in the list.

```
// Demonstrate the use of a combiner with reduce()

import java.util.*;
import java.util.stream.*;

class StreamDemo3 {

  public static void main(String[] args) {

    // This is now a list of double values.
    ArrayList<Double> myList = new ArrayList<>( );

    myList.add(7.0);
    myList.add(18.0);
    myList.add(10.0);
    myList.add(24.0);
    myList.add(17.0);
    myList.add(5.0);
```

```
    double productOfSqrRoots = myList.parallelStream().reduce(
                                      1.0,
                                      (a,b) -> a * Math.sqrt(b),
                                      (a,b) -> a * b
                                    );

    System.out.println("Product of square roots: " + productOfSqrRoots);
  }
}
```

Notice that the accumulator function multiplies the square roots of two elements, but the combiner multiplies the partial results. Thus, the two functions differ. Moreover, for this computation to work correctly, they *must* differ. For example, if you tried to obtain the product of the square roots of the elements by using the following statement, an error would result:

```
// This won't work.
double productOfSqrRoots2 = myList.parallelStream().reduce(
                                    1.0,
                                    (a,b) -> a * Math.sqrt(b));
```

In this version of **reduce( )**, the accumulator and the combiner function are one and the same. This results in an error because when two partial results are combined, their square roots are multiplied together rather than the partial results, themselves.

As a point of interest, if the stream in the preceding call to **reduce( )** had been changed to a sequential stream, then the operation would yield the correct answer because there would have been no need to combine two partial results. The problem occurs when a parallel stream is used.

You can switch a parallel stream to sequential by calling the **sequential( )** method, which is specified by **BaseStream**. It is shown here:

S sequential( )

In general, a stream can be switched between parallel and sequential on an as-needed basis.

There is one other aspect of a stream to keep in mind when using parallel execution: the order of the elements. Streams can be either ordered or unordered. In general, if the data source is ordered, then the stream will also be ordered. However, when using a parallel stream, a performance boost can sometimes be obtained by allowing a stream to be unordered. When a parallel stream is unordered, each partition of the stream can be operated on independently, without having to coordinate with the others. In cases in which the order of the operations does not matter, it is possible to specify unordered behavior by calling the **unordered( )** method, shown here:

S unordered( )

One other point: the **forEach( )** method may not preserve the ordering of a parallel stream. If you want to perform an operation on each element in a parallel stream while preserving the order, consider using **forEachOrdered( )**. It is used just like **forEach( )**.

## Mapping

Often it is useful to map the elements of one stream to another. For example, a stream that contains a database of name, telephone, and e-mail address information might map only the name and e-mail address portions to another stream. As another example, you might want to apply some transformation to the elements in a stream. To do this, you could map the transformed elements to a new stream. Because mapping operations are quite common, the stream API provides built-in support for them. The most general mapping method is **map( )**. It is shown here:

<R> Stream<R> map(Function<? super T, ? extends R> *mapFunc*)

Here, **R** specifies the type of elements of the new stream; **T** is the type of elements of the invoking stream; and *mapFunc* is an instance of **Function**, which does the mapping. The map function must be stateless and non-interfering. Since a new stream is returned, **map( )** is an intermediate method.

**Function** is a functional interface declared in **java.util.function**. It is declared as shown here:

Function<T, R>

As it relates to **map( )**, **T** is the element type and **R** is the result of the mapping. **Function** has the abstract method shown here:

R apply(T *val*)

Here, *val* is a reference to the object being mapped. The mapped result is returned.

The following is a simple example of **map( )**. It provides a variation on the previous example program. As before, the program computes the product of the square roots of the values in an **ArrayList**. In this version, however, the square roots of the elements are first mapped to a new stream. Then, **reduce( )** is employed to compute the product.

```
// Map one stream to another.

import java.util.*;
import java.util.stream.*;

class StreamDemo4 {

  public static void main(String[] args) {

    // A list of double values.
    ArrayList<Double> myList = new ArrayList<>( );

    myList.add(7.0);
    myList.add(18.0);
    myList.add(10.0);
    myList.add(24.0);
    myList.add(17.0);
    myList.add(5.0);

    // Map the square root of the elements in myList to a new stream.
    Stream<Double> sqrtRootStrm = myList.stream().map((a) -> Math.sqrt(a));
```

```
    // Find the product of the square roots.
    double productOfSqrRoots = sqrtRootStrm.reduce(1.0, (a,b) -> a*b);

    System.out.println("Product of square roots is " + productOfSqrRoots);
  }
}
```

The output is the same as before. The difference between this version and the previous is simply that the transformation (i.e., the computation of the square roots) occurs during mapping, rather than during the reduction. Because of this, it is possible to use the two-parameter form of **reduce( )** to compute the product because it is no longer necessary to provide a separate combiner function.

Here is an example that uses **map( )** to create a new stream that contains only selected fields from the original stream. In this case, the original stream contains objects of type **NamePhoneEmail**, which contains names, phone numbers, and e-mail addresses. The program then maps only the names and phone numbers to a new stream of **NamePhone** objects. The e-mail addresses are discarded.

```
// Use map() to create a new stream that contains only
// selected aspects of the original stream.

import java.util.*;
import java.util.stream.*;

class NamePhoneEmail {
  String name;
  String phonenum;
  String email;

  NamePhoneEmail(String n, String p, String e) {
    name = n;
    phonenum = p;
    email = e;
  }
}

class NamePhone {
  String name;
  String phonenum;

  NamePhone(String n, String p) {
    name = n;
    phonenum = p;
  }
}

class StreamDemo5 {

  public static void main(String[] args) {

    // A list of names, phone numbers, and e-mail addresses.
    ArrayList<NamePhoneEmail> myList = new ArrayList<>( );
```

```
    myList.add(new NamePhoneEmail("Larry", "555-5555",
                                  "Larry@HerbSchildt.com"));
    myList.add(new NamePhoneEmail("James", "555-4444",
                                  "James@HerbSchildt.com"));
    myList.add(new NamePhoneEmail("Mary", "555-3333",
                                  "Mary@HerbSchildt.com"));

    System.out.println("Original values in myList: ");
    myList.stream().forEach( (a) -> {
      System.out.println(a.name + " " + a.phonenum + " " + a.email);
    });
    System.out.println();

    // Map just the names and phone numbers to a new stream.
    Stream<NamePhone> nameAndPhone = myList.stream().map(
                                     (a) -> new NamePhone(a.name,a.phonenum)
                                    );

    System.out.println("List of names and phone numbers: ");
    nameAndPhone.forEach( (a) -> {
      System.out.println(a.name + " " + a.phonenum);
    });
  }
}
```

The output, shown here, verifies the mapping:

```
Original values in myList:
Larry 555-5555 Larry@HerbSchildt.com
James 555-4444 James@HerbSchildt.com
Mary 555-3333 Mary@HerbSchildt.com

List of names and phone numbers:
Larry 555-5555
James 555-4444
Mary 555-3333
```

Because you can pipeline more than one intermediate operation together, you can easily create very powerful actions. For example, the following statement uses **filter( )** and then **map( )** to produce a new stream that contains only the name and phone number of the elements with the name "James":

```
Stream<NamePhone> nameAndPhone = myList.stream().
                                 filter((a) -> a.name.equals("James")).
                                 map((a) -> new NamePhone(a.name,a.phonenum));
```

This type of filter operation is very common when creating database-style queries. As you gain experience with the stream API, you will find that such chains of operations can be used to create very sophisticated queries, merges, and selections on a data stream.

In addition to the version just described, three other versions of **map( )** are provided. They return a primitive stream, as shown here:

IntStream mapToInt(ToIntFunction<? super T> *mapFunc*)

LongStream mapToLong(ToLongFunction<? super T> *mapFunc*)

DoubleStream mapToDouble(ToDoubleFunction<? super T> *mapFunc*)

Each *mapFunc* must implement the abstract method defined by the specified interface, returning a value of the indicated type. For example, **ToDoubleFunction** specifies the **applyAsDouble(T *val*)** method, which must return the value of its parameter as a **double**.

Here is an example that uses a primitive stream. It first creates an **ArrayList** of **Double** values. It then uses **stream( )** followed by **mapToInt( )** to create an **IntStream** that contains the ceiling of each value.

```
// Map a Stream to an IntStream.

import java.util.*;
import java.util.stream.*;

class StreamDemo6 {

  public static void main(String[] args) {

    // A list of double values.
    ArrayList<Double> myList = new ArrayList<>( );

    myList.add(1.1);
    myList.add(3.6);
    myList.add(9.2);
    myList.add(4.7);
    myList.add(12.1);
    myList.add(5.0);

    System.out.print("Original values in myList: ");
    myList.stream().forEach( (a) -> {
      System.out.print(a + " ");
    });
    System.out.println();

    // Map the ceiling of the elements in myList to an IntStream.
    IntStream cStrm = myList.stream().mapToInt((a) -> (int) Math.ceil(a));

    System.out.print("The ceilings of the values in myList: ");
    cStrm.forEach( (a) -> {
      System.out.print(a + " ");
    });

  }
}
```

The output is shown here:

```
Original values in myList: 1.1 3.6 9.2 4.7 12.1 5.0
The ceilings of the values in myList: 2 4 10 5 13 5
```

The stream produced by **mapToInt( )** contains the ceiling values of the original elements in **myList**.

Before leaving the topic of mapping, it is necessary to point out that the stream API also provides methods that support *flat maps*. These are **flatMap( )**, **flatMapToInt( )**, **flatMapToLong( )**, and **flatMapToDouble( )**. The flat map methods are designed to handle situations in which each element in the original stream is mapped to more than one element in the resulting stream.

## Collecting

As the preceding examples have shown, it is possible (indeed, common) to obtain a stream from a collection. Sometimes it is desirable to obtain the opposite: to obtain a collection from a stream. To perform such an action, the stream API provides the **collect( )** method. It has two forms. The one we will use first is shown here:

<R, A> R collect(Collector<? super T, A, R> *collectorFunc*)

Here, **R** specifies the type of the result, and **T** specifies the element type of the invoking stream. The internal accumulated type is specified by **A**. The *collectorFunc* specifies how the collection process works. The **collect( )** method is a terminal operation.

The **Collector** interface is declared in **java.util.stream**, as shown here:

interface Collector<T, A, R>

**T**, **A**, and **R** have the same meanings as just described. **Collector** specifies several methods, but for the purposes of this chapter, we won't need to implement them. Instead, we will use two of the predefined collectors that are provided by the **Collectors** class, which is packaged in **java.util.stream**.

The **Collectors** class defines a number of static collector methods that you can use as-is. The two we will use are **toList( )** and **toSet( )**, shown here:

static <T> Collector<T, ?, List<T>> toList( )

static <T> Collector<T, ?, Set<T>> toSet( )

The **toList( )** method returns a collector that can be used to collect elements into a **List**. The **toSet( )** method returns a collector that can be used to collect elements into a **Set**. For example, to collect elements into a **List**, you can call **collect( )** like this:

```
collect(Collectors.toList())
```

The following program puts the preceding discussion into action. It reworks the example in the previous section so that it collects the names and phone numbers into a **List** and a **Set**.

```
// Use collect() to create a List and a Set from a stream.

import java.util.*;
import java.util.stream.*;

class NamePhoneEmail {
  String name;
  String phonenum;
  String email;

  NamePhoneEmail(String n, String p, String e) {
    name = n;
    phonenum = p;
    email = e;
  }
}

class NamePhone {
  String name;
  String phonenum;

  NamePhone(String n, String p) {
    name = n;
    phonenum = p;
  }
}

class StreamDemo7 {

  public static void main(String[] args) {

    // A list of names, phone numbers, and e-mail addresses.
    ArrayList<NamePhoneEmail> myList = new ArrayList<>( );

    myList.add(new NamePhoneEmail("Larry", "555-5555",
                                  "Larry@HerbSchildt.com"));
    myList.add(new NamePhoneEmail("James", "555-4444",
                                  "James@HerbSchildt.com"));
    myList.add(new NamePhoneEmail("Mary", "555-3333",
                                  "Mary@HerbSchildt.com"));

    // Map just the names and phone numbers to a new stream.
    Stream<NamePhone> nameAndPhone = myList.stream().map(
                                       (a) -> new NamePhone(a.name,a.phonenum)
                                     );

    // Use collect to create a List of the names and phone numbers.
    List<NamePhone> npList = nameAndPhone.collect(Collectors.toList());
```

```
    System.out.println("Names and phone numbers in a List:");
    for(NamePhone e : npList)
      System.out.println(e.name + ": " + e.phonenum);

    // Obtain another mapping of the names and phone numbers.
    nameAndPhone = myList.stream().map(
                                    (a) -> new NamePhone(a.name,a.phonenum)
                                   );

    // Now, create a Set by use of collect().
    Set<NamePhone> npSet = nameAndPhone.collect(Collectors.toSet());

    System.out.println("\nNames and phone numbers in a Set:");
    for(NamePhone e : npSet)
      System.out.println(e.name + ": " + e.phonenum);
  }
}
```

The output is shown here:

```
Names and phone numbers in a List:
Larry: 555-5555
James: 555-4444
Mary: 555-3333

Names and phone numbers in a Set:
James: 555-4444
Larry: 555-5555
Mary: 555-3333
```

In the program, the following line collects the name and phone numbers into a **List** by using **toList( )**:

```
List<NamePhone> npList = nameAndPhone.collect(Collectors.toList());
```

After this line executes, the collection referred to by **npList** can be used like any other **List** collection. For example, it can be cycled through by using a for-each **for** loop, as shown in the next line:

```
for(NamePhone e : npList)
  System.out.println(e.name + ": " + e.phonenum);
```

The creation of a **Set** via **collect(Collectors.toSet( ))** works in the same way. The ability to move data from a collection to a stream, and then back to a collection again is a very powerful attribute of the stream API. It gives you the ability to operate on a collection through a stream, but then repackage it as a collection. Furthermore, the stream operations can, if appropriate, occur in parallel.

The version of **collect( )** used by the previous example is quite convenient, and often the one you want, but there is a second version that gives you more control over the collection process. It is shown here:

```
<R> R collect(Supplier<R> target, BiConsumer<R, ? super T> accumulator,
              BiConsumer <R, R> combiner)
```

Here, *target* specifies how the object that holds the result is created. For example, to use a **LinkedList** as the result collection, you would specify its constructor. The *accumulator* function adds an element to the result and *combiner* combines two partial results. Thus, these functions work similarly to the way they do in **reduce( )**. For both, they must be stateless and non-interfering. They must also be associative.

Note that the *target* parameter is of type **Supplier**. It is a functional interface declared in **java.util.function**. It specifies only the **get( )** method, which has no parameters and, in this case, returns an object of type **R**. Thus, as it relates to **collect( )**, **get( )** returns a reference to a mutable storage object, such as a collection.

Note also that the types of *accumulator* and *combiner* are **BiConsumer**. This is a functional interface defined in **java.util.function**. It specifies the abstract method **accept( )** that is shown here:

void accept(T *obj*, U *obj2*)

This method performs some type of operation on *obj* and *obj2*. As it relates to *accumulator*, *obj* specifies the target collection, and *obj2* specifies the element to add to that collection. As it relates to *combiner*, *obj* and *obj2* specify two collections that will be combined.

Using the version of **collect( )** just described, you could use a **LinkedList** as the target in the preceding program, as shown here:

```
LinkedList<NamePhone> npList = nameAndPhone.collect(
                                  () -> new LinkedList<>(),
                                  (list, element) -> list.add(element),
                                  (listA,listB ) -> listA.addAll(listB));
```

Notice that the first argument to **collect( )** is a lambda expression that returns a new **LinkedList**. The second argument uses the standard collection method **add( )** to add an element to the list. The third element uses **addAll( )** to combine two linked lists. As a point of interest, you can use any method defined by **LinkedList** to add an element to the list. For example, you could use **addFirst( )** to add elements to the start of the list, as shown here:

```
(list, element) -> list.addFirst(element)
```

As you may have guessed, it is not always necessary to specify a lambda expression for the arguments to **collect( )**. Often, method and/or constructor references will suffice. For example, again assuming the preceding program, this statement creates a **HashSet** that contains all of the elements in the **nameAndPhone** stream:

```
HashSet<NamePhone> npSet = nameAndPhone.collect(HashSet::new,
                                                HashSet::add,
                                                HashSet::addAll);
```

Notice that the first argument specifies the **HashSet** constructor reference. The second and third specify method references to **HashSet**'s **add( )** and **addAll( )** methods.

One last point: In the language of the stream API, the **collect( )** method performs what is called a *mutable reduction*. This is because the result of the reduction is a mutable (i.e., changeable) storage object, such as a collection.

# Iterators and Streams

Although a stream is not a data storage object, you can still use an iterator to cycle through its elements in much the same way as you would use an iterator to cycle through the elements of a collection. The stream API supports two types of iterators. The first is the traditional **Iterator**. The second is **Spliterator**, which was added by JDK 8. It provides significant advantages in certain situations when used with parallel streams.

## Use an Iterator with a Stream

As just mentioned, you can use an iterator with a stream in just the same way that you do with a collection. Iterators are discussed in Chapter 18, but a brief review will be useful here. Iterators are objects that implement the **Iterator** interface declared in **java.util**. Its two key methods are **hasNext( )** and **next( )**. If there is another element to iterate, **hasNext( )** returns **true**, and **false** otherwise. The **next( )** method returns the next element in the iteration.

---

**NOTE** JDK 8 adds additional iterator types that handle the primitive streams: **PrimitiveIterator**, **PrimitiveIterator.OfDouble**, **PrimitiveIterator.OfLong**, and **PrimitiveIterator.OfInt**. These iterators all extend the **Iterator** interface and work in the same general way as those based directly on **Iterator**.

---

To obtain an iterator to a stream, call **iterator( )** on the stream. The version used by **Stream** is shown here.

```
Iterator<T> iterator( )
```

Here, **T** specifies the element type. (The primitive streams return iterators of the appropriate primitive type.)

The following program shows how to iterate through the elements of a stream. In this case, the strings in an **ArrayList** are iterated, but the process is the same for any type of stream.

```
// Use an iterator with a stream.

import java.util.*;
import java.util.stream.*;

class StreamDemo8 {

  public static void main(String[] args) {

    // Create a list of Strings.
    ArrayList<String> myList = new ArrayList<>( );
    myList.add("Alpha");
    myList.add("Beta");
    myList.add("Gamma");
    myList.add("Delta");
    myList.add("Phi");
    myList.add("Omega");

    // Obtain a Stream to the array list.
    Stream<String> myStream = myList.stream();
```

```
    // Obtain an iterator to the stream.
    Iterator<String> itr = myStream.iterator();

    // Iterate the elements in the stream.
    while(itr.hasNext())
      System.out.println(itr.next());
  }
}
```

The output is shown here:

```
Alpha
Beta
Gamma
Delta
Phi
Omega
```



## Use Spliterator

**Spliterator** offers an alternative to **Iterator**, especially when parallel processing is involved. In general, **Spliterator** is more sophisticated than **Iterator**, and a discussion of **Spliterator** is found in Chapter 18. However, it will be useful to review its key features here. **Spliterator** defines several methods, but we only need to use three. The first is **tryAdvance( )**. It performs an action on the next element and then advances the iterator. It is shown here:

boolean tryAdvance(Consumer<? super T> *action*)

Here, *action* specifies the action that is executed on the next element in the iteration. **tryAdvance( )** returns **true** if there is a next element. It returns **false** if no elements remain. As discussed earlier in this chapter, **Consumer** declares one method called **accept( )** that receives an element of type **T** as an argument and returns **void**.

Because **tryAdvance( )** returns **false** when there are no more elements to process, it makes the iteration loop construct very simple, for example:

```
while(splitItr.tryAdvance( // perform action here );
```

As long as **tryAdvance( )** returns **true**, the action is applied to the next element. When **tryAdvance( )** returns **false**, the iteration is complete. Notice how **tryAdvance( )** consolidates the purposes of **hasNext( )** and **next( )** provided by **Iterator** into a single method. This improves the efficiency of the iteration process.

The following version of the preceding program substitutes a **Spliterator** for the **Iterator**:

```
// Use a Spliterator.

import java.util.*;
import java.util.stream.*;

class StreamDemo9 {

  public static void main(String[] args) {
```

```
    // Create a list of Strings.
    ArrayList<String> myList = new ArrayList<>( );
    myList.add("Alpha");
    myList.add("Beta");
    myList.add("Gamma");
    myList.add("Delta");
    myList.add("Phi");
    myList.add("Omega");

    // Obtain a Stream to the array list.
    Stream<String> myStream = myList.stream();

    // Obtain a Spliterator.
    Spliterator<String> splitItr = myStream.spliterator();

    // Iterate the elements of the stream.
    while(splitItr.tryAdvance((n) -> System.out.println(n)));
  }
}
```

The output is the same as before.

In some cases, you might want to perform some action on each element collectively, rather than one at a time. To handle this type of situation, **Spliterator** provides the **forEachRemaining( )** method, shown here:

default void forEachRemaining(Consumer<? super T> *action*)

This method applies *action* to each unprocessed element and then returns. For example, assuming the preceding program, the following displays the strings remaining in the stream:

```
splitItr.forEachRemaining((n) -> System.out.println(n));
```

Notice how this method eliminates the need to provide a loop to cycle through the elements one at a time. This is another advantage of **Spliterator**.

One other **Spliterator** method of particular interest is **trySplit( )**. It splits the elements being iterated in two, returning a new **Spliterator** to one of the partitions. The other partition remains accessible by the original **Spliterator**. It is shown here:

Spliterator<T> trySplit( )

If it is not possible to split the invoking **Spliterator**, **null** is returned. Otherwise, a reference to the partition is returned. For example, here is another version of the preceding program that demonstrates **trySplit( )**:

```
// Demonstrate trySplit().

import java.util.*;
import java.util.stream.*;

class StreamDemo10 {

  public static void main(String[] args) {
```

```
    // Create a list of Strings.
    ArrayList<String> myList = new ArrayList<>( );
    myList.add("Alpha");
    myList.add("Beta");
    myList.add("Gamma");
    myList.add("Delta");
    myList.add("Phi");
    myList.add("Omega");

    // Obtain a Stream to the array list.
    Stream<String> myStream = myList.stream();

    // Obtain a Spliterator.
    Spliterator<String> splitItr = myStream.spliterator();

    // Now, split the first iterator.
    Spliterator<String> splitItr2 = splitItr.trySplit();

     // If splitItr could be split, use splitItr2 first.
    if(splitItr2 != null) {
      System.out.println("Output from splitItr2: ");
      splitItr2.forEachRemaining((n) -> System.out.println(n));
    }

    // Now, use the splitItr.
    System.out.println("\nOutput from splitItr: ");
    splitItr.forEachRemaining((n) -> System.out.println(n));
  }
}
```

The output is shown here:

```
Output from splitItr2:
Alpha
Beta
Gamma

Output from splitItr:
Delta
Phi
Omega
```

Although splitting the **Spliterator** in this simple illustration is of no practical value, splitting can be of *great value* when parallel processing over large data sets. However, in many cases, it is better to use one of the other **Stream** methods in conjunction with a parallel stream, rather than manually handling these details with **Spliterator**. **Spliterator** is primarily for the cases in which none of the predefined methods seems appropriate.

## More to Explore in the Stream API

This chapter has discussed several key aspects of the stream API and introduced the techniques required to use them, but the stream API has much more to offer. To begin, here are a few of the other methods provided by **Stream** that you will find helpful:

- To determine if one or more elements in a stream satisfy a specified predicate, use **allMatch( )**, **anyMatch( )**, or **noneMatch( )**.
- To obtain the number of elements in the stream, call **count( )**.
- To obtain a stream that contains only unique elements, use **distinct( )**.
- To create a stream that contains a specified set of elements, use **of( )**.

One last point: the stream API is a powerful addition to Java. It is likely that it will be enhanced over time to include even more functionality. Therefore, a periodic perusal of its API documentation is advised.

CHAPTER

# 30 Regular Expressions and Other Packages

When Java was originally released, it included a set of eight packages, called the *core API*. Each subsequent release added to the API. Today, the Java API contains a very large number of packages. Many of the packages support areas of specialization that are beyond the scope of this book. However, several packages warrant an examination here. Four are **java.util.regex**, **java.lang.reflect**, **java.rmi**, and **java.text**. They support regular expression processing, reflection, Remote Method Invocation (RMI), and text formatting, respectively. The chapter ends by introducing the new date and time API in **java.time** and its subpackages.

The *regular expression* package lets you perform sophisticated pattern matching operations. This chapter provides an in-depth introduction to this package along with extensive examples. *Reflection* is the ability of software to analyze itself. It is an essential part of the Java Beans technology that is covered in Chapter 37. *Remote Method Invocation (RMI)* allows you to build Java applications that are distributed among several machines. This chapter provides a simple client/server example that uses RMI. The *text formatting* capabilities of **java.text** have many uses. The one examined here formats date and time strings. The new date and time API supplies an up-to-date approach to handling date and time.

## The Core Java API Packages

At the time of this writing, Table 30-1 lists all of the core API packages defined by Java (those in the **java** namespace) and summarizes their functions.

| Package | Primary Function |
|---|---|
| java.applet | Supports construction of applets. |
| java.awt | Provides capabilities for graphical user interfaces. |
| java.awt.color | Supports color spaces and profiles. |
| java.awt.datatransfer | Transfers data to and from the system clipboard. |

**Table 30-1** The Core Java API Packages

| Package | Primary Function |
|---|---|
| java.awt.dnd | Supports drag-and-drop operations. |
| java.awt.event | Handles events. |
| java.awt.font | Represents various types of fonts. |
| java.awt.geom | Allows you to work with geometric shapes. |
| java.awt.im | Allows input of Japanese, Chinese, and Korean characters to text editing components. |
| java.awt.im.spi | Supports alternative input devices. |
| java.awt.image | Processes images. |
| java.awt.image.renderable | Supports rendering-independent images. |
| java.awt.print | Supports general print capabilities. |
| java.beans | Allows you to build software components. |
| java.beans.beancontext | Provides an execution environment for Beans. |
| java.io | Inputs and outputs data. |
| java.lang | Provides core functionality. |
| java.lang.annotation | Supports annotations (metadata). |
| java.lang.instrument | Supports program instrumentation. |
| java.lang.invoke | Supports dynamic languages. |
| java.lang.management | Supports management of the execution environment. |
| java.lang.ref | Enables some interaction with the garbage collector. |
| java.lang.reflect | Analyzes code at run time. |
| java.math | Handles large integers and decimal numbers. |
| java.net | Supports networking. |
| java.nio | Top-level package for the NIO classes. Encapsulates buffers. |
| java.nio.channels | Encapsulates channels, which are used by the NIO system. |
| java.nio.channels.spi | Supports service providers for channels. |
| java.nio.charset | Encapsulates character sets. |
| java.nio.charset.spi | Supports service providers for character sets. |
| java.nio.file | Provides NIO support for files. |
| java.nio.file.attribute | Supports NIO file attributes. |
| java.nio.file.spi | Supports NIO service providers for files. |
| java.rmi | Provides remote method invocation. |
| java.rmi.activation | Activates persistent objects. |
| java.rmi.dgc | Manages distributed garbage collection. |
| java.rmi.registry | Maps names to remote object references. |
| java.rmi.server | Supports remote method invocation. |
| java.security | Handles certificates, keys, digests, signatures, and other security functions. |

**Table 30-1** The Core Java API Packages *(continued)*

| Package | Primary Function |
|---|---|
| java.security.acl | Manages access control lists. |
| java.security.cert | Parses and manages certificates. |
| java.security.interfaces | Defines interfaces for DSA (Digital Signature Algorithm) keys. |
| java.security.spec | Specifies keys and algorithm parameters. |
| java.sql | Communicates with a SQL (Structured Query Language) database. |
| java.text | Formats, searches, and manipulates text. |
| java.text.spi | Supports service providers for text formatting classes in **java.text**. |
| java.time | Primary support for the new date and time API. (Added by JDK 8.) |
| java.time.chrono | Supports alternative, non-Gregorian calendars. (Added by JDK 8.) |
| java.time.format | Supports date and time formatting. (Added by JDK 8.) |
| java.time.temporal | Supports extended date and time functionality. (Added by JDK 8.) |
| java.time.zone | Supports time zones. (Added by JDK 8.) |
| java.util | Contains common utilities. |
| java.util.concurrent | Supports the concurrent utilities. |
| java.util.concurrent.atomic | Supports atomic (that is, indivisible) operations on variables without the use of locks. |
| java.util.concurrent.locks | Supports synchronization locks. |
| java.util.function | Provides several functional interfaces. (Added by JDK 8.) |
| java.util.jar | Creates and reads JAR files. |
| java.util.logging | Supports logging of information related to a program's execution. |
| java.util.prefs | Encapsulates information relating to user preference. |
| java.util.regex | Supports regular expression processing. |
| java.util.spi | Supports service providers for the utility classes in **java.util**. |
| java.util.stream | Supports the new stream API. (Added by JDK 8.) |
| java.util.zip | Reads and writes compressed and uncompressed ZIP files. |

**Table 30-1** The Core Java API Packages *(continued)*



## Regular Expression Processing

The **java.util.regex** package supports regular expression processing. As the term is used here, a *regular expression* is a string of characters that describes a character sequence. This general description, called a *pattern*, can then be used to find matches in other character sequences. Regular expressions can specify wildcard characters, sets of characters, and various quantifiers. Thus, you can specify a regular expression that represents a general form that can match several different specific character sequences.

There are two classes that support regular expression processing: **Pattern** and **Matcher**. These classes work together. Use **Pattern** to define a regular expression. Match the pattern against another sequence using **Matcher**.

## Pattern

The **Pattern** class defines no constructors. Instead, a pattern is created by calling the **compile( )** factory method. One of its forms is shown here:

static Pattern compile(String *pattern*)

Here, *pattern* is the regular expression that you want to use. The **compile( )** method transforms the string in *pattern* into a pattern that can be used for pattern matching by the **Matcher** class. It returns a **Pattern** object that contains the pattern.

Once you have created a **Pattern** object, you will use it to create a **Matcher**. This is done by calling the **matcher( )** factory method defined by **Pattern**. It is shown here:

Matcher matcher(CharSequence *str*)

Here *str* is the character sequence that the pattern will be matched against. This is called the *input sequence*. **CharSequence** is an interface that defines a read-only set of characters. It is implemented by the **String** class, among others. Thus, you can pass a string to **matcher( )**.

## Matcher

The **Matcher** class has no constructors. Instead, you create a **Matcher** by calling the **matcher( )** factory method defined by **Pattern**, as just explained. Once you have created a **Matcher**, you will use its methods to perform various pattern matching operations.

The simplest pattern matching method is **matches( )**, which simply determines whether the character sequence matches the pattern. It is shown here:

boolean matches( )

It returns **true** if the sequence and the pattern match, and **false** otherwise. Understand that the entire sequence must match the pattern, not just a subsequence of it.

To determine if a subsequence of the input sequence matches the pattern, use **find( )**. One version is shown here:

boolean find( )

It returns **true** if there is a matching subsequence and **false** otherwise. This method can be called repeatedly, allowing it to find all matching subsequences. Each call to **find( )** begins where the previous one left off.

You can obtain a string containing the last matching sequence by calling **group( )**. One of its forms is shown here:

String group( )

The matching string is returned. If no match exists, then an **IllegalStateException** is thrown.

You can obtain the index within the input sequence of the current match by calling **start( )**. The index one past the end of the current match is obtained by calling **end( )**. The forms used in this chapter are shown here:

int start( )
int end( )

Both throw **IllegalStateException** if no match exists.

You can replace all occurrences of a matching sequence with another sequence by calling **replaceAll( )**, shown here:

String replaceAll(String *newStr*)

Here, *newStr* specifies the new character sequence that will replace the ones that match the pattern. The updated input sequence is returned as a string.

## Regular Expression Syntax

Before demonstrating **Pattern** and **Matcher**, it is necessary to explain how to construct a regular expression. Although no rule is complicated by itself, there are a large number of them, and a complete discussion is beyond the scope of this chapter. However, a few of the more commonly used constructs are described here.

In general, a regular expression is comprised of normal characters, character classes (sets of characters), wildcard characters, and quantifiers. A normal character is matched as-is. Thus, if a pattern consists of "xy", then the only input sequence that will match it is "xy". Characters such as newline and tab are specified using the standard escape sequences, which begin with a \. For example, a newline is specified by **\n**. In the language of regular expressions, a normal character is also called a *literal*.

A character class is a set of characters. A character class is specified by putting the characters in the class between brackets. For example, the class [wxyz] matches w, x, y, or z. To specify an inverted set, precede the characters with a ^. For example, [^wxyz] matches any character except w, x, y, or z. You can specify a range of characters using a hyphen. For example, to specify a character class that will match the digits 1 through 9, use [1-9].

The wildcard character is the **.** (dot) and it matches any character. Thus, a pattern that consists of "." will match these (and other) input sequences: "A", "a", "x", and so on.

A quantifier determines how many times an expression is matched. The quantifiers are shown here:

| | |
|---|---|
| + | Match one or more. |
| * | Match zero or more. |
| ? | Match zero or one. |

For example, the pattern "x+" will match "x", "xx", and "xxx", among others.

One other point: In general, if you specify an invalid expression, a **PatternSyntaxException** will be thrown.

## Demonstrating Pattern Matching

The best way to understand how regular expression pattern matching operates is to work through some examples. The first, shown here, looks for a match with a literal pattern:

```
// A simple pattern matching demo.
import java.util.regex.*;

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

```
    Pattern pat;
    Matcher mat;
    boolean found;

    pat = Pattern.compile("Java");
    mat = pat.matcher("Java");
    found = mat.matches(); // check for a match

    System.out.println("Testing Java against Java.");
    if(found) System.out.println("Matches");
    else System.out.println("No Match");

    System.out.println();

    System.out.println("Testing Java against Java 8.");
    mat = pat.matcher("Java 8"); // create a new matcher

    found = mat.matches(); // check for a match

    if(found) System.out.println("Matches");
    else System.out.println("No Match");
  }
}
```

The output from the program is shown here:

```
Testing Java against Java.
Matches

Testing Java against Java 8.
No Match
```

Let's look closely at this program. The program begins by creating the pattern that contains the sequence "Java". Next, a **Matcher** is created for that pattern that has the input sequence "Java". Then, the **matches( )** method is called to determine if the input sequence matches the pattern. Because the sequence and the pattern are the same, **matches( )** returns **true**. Next, a new **Matcher** is created with the input sequence "Java 8" and **matches( )** is called again. In this case, the pattern and the input sequence differ, and no match is found. Remember, the **matches( )** function returns **true** only when the input sequence precisely matches the pattern. It will not return **true** just because a subsequence matches.

You can use **find( )** to determine if the input sequence contains a subsequence that matches the pattern. Consider the following program:

```
// Use find() to find a subsequence.
import java.util.regex.*;

class RegExpr2 {
  public static void main(String args[]) {
    Pattern pat = Pattern.compile("Java");
    Matcher mat = pat.matcher("Java 8");

    System.out.println("Looking for Java in Java 8.");
```

```
    if(mat.find()) System.out.println("subsequence found");
    else System.out.println("No Match");
  }
}
```

The output is shown here:

```
Looking for Java in Java 8.
subsequence found
```

In this case, **find( )** finds the subsequence "Java".

The **find( )** method can be used to search the input sequence for repeated occurrences of the pattern because each call to **find( )** picks up where the previous one left off. For example, the following program finds two occurrences of the pattern "test":

```
// Use find() to find multiple subsequences.
import java.util.regex.*;

class RegExpr3 {
  public static void main(String args[]) {
    Pattern pat = Pattern.compile("test");
    Matcher mat = pat.matcher("test 1 2 3 test");

    while(mat.find()) {
      System.out.println("test found at index " +
                         mat.start());
    }
  }
}
```

The output is shown here:

```
test found at index 0
test found at index 11
```

As the output shows, two matches were found. The program uses the **start( )** method to obtain the index of each match.

## Using Wildcards and Quantifiers

Although the preceding programs show the general technique for using **Pattern** and **Matcher**, they don't show their power. The real benefit of regular expression processing is not seen until wildcards and quantifiers are used. To begin, consider the following example that uses the + quantifier to match any arbitrarily long sequence of Ws:

```
// Use a quantifier.
import java.util.regex.*;

class RegExpr4 {
  public static void main(String args[]) {
    Pattern pat = Pattern.compile("W+");
    Matcher mat = pat.matcher("W WW WWW");
```

```
    while(mat.find())
      System.out.println("Match: " + mat.group());
  }
}
```

The output from the program is shown here:

```
Match: W
Match: WW
Match: WWW
```

As the output shows, the regular expression pattern "W+" matches any arbitrarily long sequence of Ws.

The next program uses a wildcard to create a pattern that will match any sequence that begins with *e* and ends with *d*. To do this, it uses the dot wildcard character along with the + quantifier.

```
// Use wildcard and quantifier.
import java.util.regex.*;

class RegExpr5 {
  public static void main(String args[]) {
    Pattern pat = Pattern.compile("e.+d");
    Matcher mat = pat.matcher("extend cup end table");

    while(mat.find())
      System.out.println("Match: " + mat.group());
  }
}
```

You might be surprised by the output produced by the program, which is shown here:

```
Match: extend cup end
```

Only one match is found, and it is the longest sequence that begins with *e* and ends with *d*. You might have expected two matches: "extend" and "end". The reason that the longer sequence is found is that, by default, **find( )** matches the longest sequence that fits the pattern. This is called *greedy behavior*. You can specify *reluctant behavior* by adding the **?** quantifier to the pattern, as shown in this version of the program. It causes the shortest matching pattern to be obtained.

```
// Use the ? quantifier.
import java.util.regex.*;

class RegExpr6 {
  public static void main(String args[]) {
    // Use reluctant matching behavior.
    Pattern pat = Pattern.compile("e.+?d");
    Matcher mat = pat.matcher("extend cup end table");

    while(mat.find())
      System.out.println("Match: " + mat.group());
  }
}
```

The output from the program is shown here:

```
Match: extend
Match: end
```

As the output shows, the pattern "e.+?d" will match the shortest sequence that begins with *e* and ends with *d*. Thus, two matches are found.

## Working with Classes of Characters

Sometimes you will want to match any sequence that contains one or more characters, in any order, that are part of a set of characters. For example, to match whole words, you want to match any sequence of the letters of the alphabet. One of the easiest ways to do this is to use a character class, which defines a set of characters. Recall that a character class is created by putting the characters you want to match between brackets. For example, to match the lowercase characters a through z, use **[a-z]**. The following program demonstrates this technique:

```
// Use a character class.
import java.util.regex.*;

class RegExpr7 {
  public static void main(String args[]) {
    // Match lowercase words.
    Pattern pat = Pattern.compile("[a-z]+");
    Matcher mat = pat.matcher("this is a test.");

    while(mat.find())
      System.out.println("Match: " + mat.group());
  }
}
```

The output is shown here:

```
Match: this
Match: is
Match: a
Match: test
```

## Using replaceAll( )

The **replaceAll( )** method supplied by **Matcher** lets you perform powerful search and replace operations that use regular expressions. For example, the following program replaces all occurrences of sequences that begin with "Jon" with "Eric":

```
// Use replaceAll().
import java.util.regex.*;

class RegExpr8 {
  public static void main(String args[]) {
    String str = "Jon Jonathan Frank Ken Todd";

    Pattern pat = Pattern.compile("Jon.*? ");
    Matcher mat = pat.matcher(str);
```

```
    System.out.println("Original sequence: " + str);

    str = mat.replaceAll("Eric ");

    System.out.println("Modified sequence: " + str);

  }
}
```

The output is shown here:

```
Original sequence: Jon Jonathan Frank Ken Todd
Modified sequence: Eric Eric Frank Ken Todd
```

Because the regular expression "Jon.*? " matches any string that begins with Jon followed by zero or more characters, ending in a space, it can be used to match and replace both Jon and Jonathan with the name Eric. Such a substitution is not easily accomplished without pattern matching capabilities.

## Using split( )

You can reduce an input sequence into its individual tokens by using the **split( )** method defined by **Pattern**. One form of the **split( )** method is shown here:

String[ ] split(CharSequence *str*)

It processes the input sequence passed in *str*, reducing it into tokens based on the delimiters specified by the pattern.

For example, the following program finds tokens that are separated by spaces, commas, periods, and exclamation points:

```
// Use split().
import java.util.regex.*;

class RegExpr9 {
  public static void main(String args[]) {

    // Match lowercase words.
    Pattern pat = Pattern.compile("[ ,.!]");

    String strs[] = pat.split("one two,alpha9 12!done.");

    for(int i=0; i < strs.length; i++)
      System.out.println("Next token: " + strs[i]);

  }
}
```

The output is shown here:

```
Next token: one
Next token: two
Next token: alpha9
```

```
Next token: 12
Next token: done
```

As the output shows, the input sequence is reduced to its individual tokens. Notice that the delimiters are not included.

### Two Pattern-Matching Options

Although the pattern-matching techniques described in the foregoing offer the greatest flexibility and power, there are two alternatives which you might find useful in some circumstances. If you only need to perform a one-time pattern match, you can use the **matches( )** method defined by **Pattern**. It is shown here:

static boolean matches(String *pattern*, CharSequence *str*)

It returns **true** if *pattern* matches *str* and **false** otherwise. This method automatically compiles *pattern* and then looks for a match. If you will be using the same pattern repeatedly, then using **matches( )** is less efficient than compiling the pattern and using the pattern-matching methods defined by **Matcher**, as described previously.

You can also perform a pattern match by using the **matches( )** method implemented by **String**. It is shown here:

boolean matches(String *pattern*)

If the invoking string matches the regular expression in *pattern*, then **matches( )** returns **true**. Otherwise, it returns **false**.

### Exploring Regular Expressions

The overview of regular expressions presented in this section only hints at their power. Since text parsing, manipulation, and tokenization are a large part of programming, you will likely find Java's regular expression subsystem a powerful tool that you can use to your advantage. It is, therefore, wise to explore the capabilities of regular expressions. Experiment with several different types of patterns and input sequences. Once you understand how regular expression pattern matching works, you will find it useful in many of your programming endeavors.

## Reflection

Reflection is the ability of software to analyze itself. This is provided by the **java.lang.reflect** package and elements in **Class**. Reflection is an important capability, especially when using components called Java Beans. It allows you to analyze a software component and describe its capabilities dynamically, at run time rather than at compile time. For example, by using reflection, you can determine what methods, constructors, and fields a class supports. Reflection was introduced in Chapter 12. It is examined further here.

The package **java.lang.reflect** includes several interfaces. Of special interest is **Member**, which defines methods that allow you to get information about a field, constructor, or method of a class. There are also ten classes in this package. These are listed in Table 30-2.

| Class | Primary Function |
|---|---|
| AccessibleObject | Allows you to bypass the default access control checks. |
| Array | Allows you to dynamically create and manipulate arrays. |
| Constructor | Provides information about a constructor. |
| Executable | An abstract superclass extended by **Method** and **Constructor**. (Added by JDK 8.) |
| Field | Provides information about a field. |
| Method | Provides information about a method. |
| Modifier | Provides information about class and member access modifiers. |
| Parameter | Provides information about parameters. (Added by JDK 8.) |
| Proxy | Supports dynamic proxy classes. |
| ReflectPermission | Allows reflection of private or protected members of a class. |

**Table 30-2** Classes Defined in **java.lang.reflect**

The following application illustrates a simple use of the Java reflection capabilities. It prints the constructors, fields, and methods of the class **java.awt.Dimension**. The program begins by using the **forName( )** method of **Class** to get a class object for **java.awt.Dimension**. Once this is obtained, **getConstructors( )**, **getFields( )**, and **getMethods( )** are used to analyze this class object. They return arrays of **Constructor**, **Field**, and **Method** objects that provide the information about the object. The **Constructor**, **Field**, and **Method** classes define several methods that can be used to obtain information about an object. You will want to explore these on your own. However, each supports the **toString( )** method. Therefore, using **Constructor**, **Field**, and **Method** objects as arguments to the **println( )** method is straightforward, as shown in the program.

```
// Demonstrate reflection.
import java.lang.reflect.*;
public class ReflectionDemo1 {
  public static void main(String args[]) {
    try {
      Class<?> c = Class.forName("java.awt.Dimension");
      System.out.println("Constructors:");
      Constructor<?> constructors[] = c.getConstructors();
      for(int i = 0; i < constructors.length; i++) {
        System.out.println(" " + constructors[i]);
      }

      System.out.println("Fields:");
      Field fields[] = c.getFields();
      for(int i = 0; i < fields.length; i++) {
        System.out.println(" " + fields[i]);
      }

      System.out.println("Methods:");
      Method methods[] = c.getMethods();
```

```
      for(int i = 0; i < methods.length; i++) {
        System.out.println(" " + methods[i]);
      }
    }
    catch(Exception e) {
      System.out.println("Exception: " + e);
    }
  }
}
```

Here is the output from this program. (The precise order may differ slightly from that shown.)

```
Constructors:
 public java.awt.Dimension(int,int)
 public java.awt.Dimension()
 public java.awt.Dimension(java.awt.Dimension)
Fields:
 public int java.awt.Dimension.width
 public int java.awt.Dimension.height
Methods:
 public int java.awt.Dimension.hashCode()
 public boolean java.awt.Dimension.equals(java.lang.Object)
 public java.lang.String java.awt.Dimension.toString()
 public java.awt.Dimension java.awt.Dimension.getSize()
 public void java.awt.Dimension.setSize(double,double)
 public void java.awt.Dimension.setSize(java.awt.Dimension)
 public void java.awt.Dimension.setSize(int,int)
 public double java.awt.Dimension.getHeight()
 public double java.awt.Dimension.getWidth()
 public java.lang.Object java.awt.geom.Dimension2D.clone()
 public void java.awt.geom.
             Dimension2D.setSize(java.awt.geom.Dimension2D)
 public final native java.lang.Class java.lang.Object.getClass()
 public final native void java.lang.Object.wait(long)
   throws java.lang.InterruptedException
 public final void java.lang.Object.wait()
   throws java.lang.InterruptedException
 public final void java.lang.Object.wait(long,int)
   throws java.lang.InterruptedException
 public final native void java.lang.Object.notify()
 public final native void java.lang.Object.notifyAll()
```

The next example uses Java's reflection capabilities to obtain the public methods of a class. The program begins by instantiating class **A**. The **getClass( )** method is applied to this object reference, and it returns the **Class** object for class **A**. The **getDeclaredMethods( )** method returns an array of **Method** objects that describe only the methods declared by this class. Methods inherited from superclasses such as **Object** are not included.

Each element of the **methods** array is then processed. The **getModifiers( )** method returns an **int** containing flags that describe which modifiers apply for this element. The **Modifier** class provides a set of **is*X*** methods, shown in Table 30-3, that can be used to examine this value. For example, the static method **isPublic( )** returns **true** if its argument

| Method | Description |
|---|---|
| static boolean isAbstract(int *val*) | Returns **true** if *val* has the **abstract** flag set and **false** otherwise. |
| static boolean isFinal(int *val*) | Returns **true** if *val* has the **final** flag set and **false** otherwise. |
| static boolean isInterface(int *val*) | Returns **true** if *val* has the **interface** flag set and **false** otherwise. |
| static boolean isNative(int *val*) | Returns **true** if *val* has the **native** flag set and **false** otherwise. |
| static boolean isPrivate(int *val*) | Returns **true** if *val* has the **private** flag set and **false** otherwise. |
| static boolean isProtected(int *val*) | Returns **true** if *val* has the **protected** flag set and **false** otherwise. |
| static boolean isPublic(int *val*) | Returns **true** if *val* has the **public** flag set and **false** otherwise. |
| static boolean isStatic(int *val*) | Returns **true** if *val* has the **static** flag set and **false** otherwise. |
| static boolean isStrict(int *val*) | Returns **true** if *val* has the **strict** flag set and **false** otherwise. |
| static boolean isSynchronized(int *val*) | Returns **true** if *val* has the **synchronized** flag set and **false** otherwise. |
| static boolean isTransient(int *val*) | Returns **true** if *val* has the **transient** flag set and **false** otherwise. |
| static boolean isVolatile(int *val*) | Returns **true** if *val* has the **volatile** flag set and **false** otherwise. |

**Table 30-3** The "is" Methods Defined by **Modifier** That Determine Modifiers

includes the **public** modifier. Otherwise, it returns **false**. In the following program, if the method supports public access, its name is obtained by the **getName( )** method and is then printed.

```
// Show public methods.
import java.lang.reflect.*;
public class ReflectionDemo2 {
  public static void main(String args[]) {

    try {
      A a = new A();
      Class<?> c = a.getClass();
      System.out.println("Public Methods:");
      Method methods[] = c.getDeclaredMethods();
      for(int i = 0; i < methods.length; i++) {
        int modifiers = methods[i].getModifiers();
        if(Modifier.isPublic(modifiers)) {
```

```
            System.out.println(" " + methods[i].getName());
          }
        }
      }
      catch(Exception e) {
        System.out.println("Exception: " + e);
      }
    }
}

class A {
  public void a1() {
  }
  public void a2() {
  }
  protected void a3() {
  }
  private void a4() {
  }
}
```

Here is the output from this program:

```
Public Methods:
 a1
 a2
```

**Modifier** also includes a set of static methods that return the type of modifiers that can be applied to a specific type of program element. These methods are

static int classModifiers( )

static int constructorModifiers( )

static int fieldModifiers( )

static int interfaceModifiers( )

static int methodModifiers( )

static int parameterModifiers( ) (Added by JDK 8.)

For example, **methodModifiers( )** returns the modifiers that can be applied to a method. Each method returns flags, packed into an **int**, that indicate which modifiers are legal. The modifier values are defined by constants in **Modifier**, which include **PROTECTED**, **PUBLIC**, **PRIVATE**, **STATIC**, **FINAL**, and so on.

## Remote Method Invocation (RMI)

Remote Method Invocation (RMI) allows a Java object that executes on one machine to invoke a method of a Java object that executes on another machine. This is an important feature, because it allows you to build distributed applications. While a complete discussion of RMI is outside the scope of this book, the following simplified example describes the basic principles involved.

## A Simple Client/Server Application Using RMI

This section provides step-by-step directions for building a simple client/server application by using RMI. The server receives a request from a client, processes it, and returns a result. In this example, the request specifies two numbers. The server adds these together and returns the sum.

### Step One: Enter and Compile the Source Code

This application uses four source files. The first file, **AddServerIntf.java**, defines the remote interface that is provided by the server. It contains one method that accepts two **double** arguments and returns their sum. All remote interfaces must extend the **Remote** interface, which is part of **java.rmi**. **Remote** defines no members. Its purpose is simply to indicate that an interface uses remote methods. All remote methods can throw a **RemoteException**.

```
import java.rmi.*;

public interface AddServerIntf extends Remote {
  double add(double d1, double d2) throws RemoteException;
}
```

The second source file, **AddServerImpl.java**, implements the remote interface. The implementation of the **add( )** method is straightforward. Remote objects typically extend **UnicastRemoteObject**, which provides functionality that is needed to make objects available from remote machines.

```
import java.rmi.*;
import java.rmi.server.*;

public class AddServerImpl extends UnicastRemoteObject
  implements AddServerIntf {

  public AddServerImpl() throws RemoteException {
  }
  public double add(double d1, double d2) throws RemoteException {
    return d1 + d2;
  }
}
```

The third source file, **AddServer.java**, contains the main program for the server machine. Its primary function is to update the RMI registry on that machine. This is done by using the **rebind( )** method of the **Naming** class (found in **java.rmi**). That method associates a name with an object reference. The first argument to the **rebind( )** method is a string that names the server as "AddServer". Its second argument is a reference to an instance of **AddServerImpl**.

```
import java.net.*;
import java.rmi.*;

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

```
    try {
      AddServerImpl addServerImpl = new AddServerImpl();
      Naming.rebind("AddServer", addServerImpl);
    }
    catch(Exception e) {
      System.out.println("Exception: " + e);
    }
  }
}
```

The fourth source file, **AddClient.java**, implements the client side of this distributed application. **AddClient.java** requires three command-line arguments. The first is the IP address or name of the server machine. The second and third arguments are the two numbers that are to be summed.

The application begins by forming a string that follows the URL syntax. This URL uses the **rmi** protocol. The string includes the IP address or name of the server and the string "AddServer". The program then invokes the **lookup( )** method of the **Naming** class. This method accepts one argument, the **rmi** URL, and returns a reference to an object of type **AddServerIntf**. All remote method invocations can then be directed to this object.

The program continues by displaying its arguments and then invokes the remote **add( )** method. The sum is returned from this method and is then printed.

```
import java.rmi.*;

public class AddClient {
  public static void main(String args[]) {
    try {
      String addServerURL = "rmi://" + args[0] + "/AddServer";
      AddServerIntf addServerIntf =
                    (AddServerIntf)Naming.lookup(addServerURL);
      System.out.println("The first number is: " + args[1]);
      double d1 = Double.valueOf(args[1]).doubleValue();
      System.out.println("The second number is: " + args[2]);

      double d2 = Double.valueOf(args[2]).doubleValue();
      System.out.println("The sum is: " + addServerIntf.add(d1, d2));
    }
    catch(Exception e) {
      System.out.println("Exception: " + e);
    }
  }
}
```

After you enter all the code, use **javac** to compile the four source files that you created.

## Step Two: Manually Generate a Stub if Required

In the context of RMI, a *stub* is a Java object that resides on the client machine. Its function is to present the same interfaces as the remote server. Remote method calls initiated by the client are actually directed to the stub. The stub works with the other parts of the RMI system to formulate a request that is sent to the remote machine.

A remote method may accept arguments that are simple types or objects. In the latter case, the object may have references to other objects. All of this information must be sent to

the remote machine. That is, an object passed as an argument to a remote method call must be serialized and sent to the remote machine. Recall from Chapter 20 that the serialization facilities also recursively process all referenced objects.

If a response must be returned to the client, the process works in reverse. Note that the serialization and deserialization facilities are also used if objects are returned to a client.

Prior to Java 5, stubs needed to be built manually by using **rmic**. This step is not required for modern versions of Java. However, if you are working in a legacy environment, then you can use the **rmic** compiler, as shown here, to build a stub:

```
rmic AddServerImpl
```

This command generates the file **AddServerImpl_Stub.class**. When using **rmic**, be sure that **CLASSPATH** is set to include the current directory.

### Step Three: Install Files on the Client and Server Machines

Copy **AddClient.class**, **AddServerImpl_Stub.class** (if needed), and **AddServerIntf.class** to a directory on the client machine. Copy **AddServerIntf.class**, **AddServerImpl.class**, **AddServerImpl_Stub.class** (if needed), and **AddServer.class** to a directory on the server machine.

---

**NOTE** RMI has techniques for dynamic class loading, but they are not used by the example at hand. Instead, all of the files that are used by the client and server applications must be installed manually on those machines.

### Step Four: Start the RMI Registry on the Server Machine

The JDK provides a program called **rmiregistry**, which executes on the server machine. It maps names to object references. First, check that the **CLASSPATH** environment variable includes the directory in which your files are located. Then, start the RMI Registry from the command line, as shown here:

```
start rmiregistry
```

When this command returns, you should see that a new window has been created. You need to leave this window open until you are done experimenting with the RMI example.

### Step Five: Start the Server

The server code is started from the command line, as shown here:

```
java AddServer
```

Recall that the **AddServer** code instantiates **AddServerImpl** and registers that object with the name "AddServer".

### Step Six: Start the Client

The **AddClient** software requires three arguments: the name or IP address of the server machine and the two numbers that are to be summed together. You may invoke it from the command line by using one of the two formats shown here:

```
java AddClient server1 8 9
java AddClient 11.12.13.14 8 9
```

In the first line, the name of the server is provided. The second line uses its IP address (11.12.13.14).

You can try this example without actually having a remote server. To do so, simply install all of the programs on the same machine, start **rmiregistry**, start **AddServer**, and then execute **AddClient** using this command line:

```
java AddClient 127.0.0.1 8 9
```

Here, the address 127.0.0.1 is the "loop back" address for the local machine. Using this address allows you to exercise the entire RMI mechanism without actually having to install the server on a remote computer. (If you are using a firewall, then this approach may not work.)

In either case, sample output from this program is shown here:

```
The first number is: 8
The second number is: 9
The sum is: 17.0
```

**NOTE** When working with RMI in the real world, it may be necessary for the server to install a security manager.



# Formatting Date and Time with java.text

The package **java.text** allows you to format, parse, search, and manipulate text. This section examines two of its most commonly used classes: those that format date and time information. However, it is important to state at the outset that the new date and time API described later in this chapter offers a modern approach to handling date and time that also supports formatting. Of course, legacy code will continue to use the classes shown here for some time.

## DateFormat Class

**DateFormat** is an abstract class that provides the ability to format and parse dates and times. The **getDateInstance( )** method returns an instance of **DateFormat** that can format date information. It is available in these forms:

```
static final DateFormat getDateInstance( )
static final DateFormat getDateInstance(int style)
static final DateFormat getDateInstance(int style, Locale locale)
```

The argument *style* is one of the following values: **DEFAULT**, **SHORT**, **MEDIUM**, **LONG**, or **FULL**. These are **int** constants defined by **DateFormat**. They cause different details about the date to be presented. The argument *locale* is one of the static references defined by **Locale** (refer to Chapter 19 for details). If the *style* and/or *locale* is not specified, defaults are used.

One of the most commonly used methods in this class is **format( )**. It has several overloaded forms, one of which is shown here:

```
final String format(Date d)
```

The argument is a **Date** object that is to be displayed. The method returns a string containing the formatted information.

The following listing illustrates how to format date information. It begins by creating a **Date** object. This captures the current date and time information. Then it outputs the date information by using different styles and locales.

```
// Demonstrate date formats.
import java.text.*;
import java.util.*;

public class DateFormatDemo {
  public static void main(String args[]) {
    Date date = new Date();
    DateFormat df;

    df = DateFormat.getDateInstance(DateFormat.SHORT, Locale.JAPAN);
    System.out.println("Japan: " + df.format(date));

    df = DateFormat.getDateInstance(DateFormat.MEDIUM, Locale.KOREA);
    System.out.println("Korea: " + df.format(date));

    df = DateFormat.getDateInstance(DateFormat.LONG, Locale.UK);
    System.out.println("United Kingdom: " + df.format(date));

    df = DateFormat.getDateInstance(DateFormat.FULL, Locale.US);
    System.out.println("United States: " + df.format(date));
  }
}
```

Sample output from this program is shown here:

```
Japan: 14/01/01
Korea: 2014. 1. 1
United Kingdom: 01 January 2014
United States: Wednesday, January 1, 2014
```

The **getTimeInstance( )** method returns an instance of **DateFormat** that can format time information. It is available in these versions:

static final DateFormat getTimeInstance( )
static final DateFormat getTimeInstance(int *style*)
static final DateFormat getTimeInstance(int *style*, Locale *locale*)

The argument *style* is one of the following values: **DEFAULT**, **SHORT**, **MEDIUM**, **LONG**, or **FULL**. These are **int** constants defined by **DateFormat**. They cause different details about the time to be presented. The argument *locale* is one of the static references defined by **Locale**. If the *style* and/or *locale* is not specified, defaults are used.

The following listing illustrates how to format time information. It begins by creating a **Date** object. This captures the current date and time information. Then it outputs the time information by using different styles and locales.

```
// Demonstrate time formats.
import java.text.*;
import java.util.*;
```

```
public class TimeFormatDemo {
  public static void main(String args[]) {
    Date date = new Date();
    DateFormat df;

    df = DateFormat.getTimeInstance(DateFormat.SHORT, Locale.JAPAN);
    System.out.println("Japan: " + df.format(date));

    df = DateFormat.getTimeInstance(DateFormat.LONG, Locale.UK);
    System.out.println("United Kingdom: " + df.format(date));

    df = DateFormat.getTimeInstance(DateFormat.FULL, Locale.CANADA);
    System.out.println("Canada: " + df.format(date));
  }
}
```

Sample output from this program is shown here:

```
Japan: 13:06
United Kingdom: 13:06:53 CST
Canada: 1:06:53 o'clock PM CST
```

The **DateFormat** class also has a **getDateTimeInstance( )** method that can format both date and time information. You may wish to experiment with it on your own.

## SimpleDateFormat Class

**SimpleDateFormat** is a concrete subclass of **DateFormat**. It allows you to define your own formatting patterns that are used to display date and time information.

One of its constructors is shown here:

SimpleDateFormat(String *formatString*)

The argument *formatString* describes how date and time information is displayed. An example of its use is given here:

```
SimpleDateFormat sdf = SimpleDateFormat("dd MMM yyyy hh:mm:ss zzz");
```

The symbols used in the formatting string determine the information that is displayed. Table 30-4 lists these symbols and gives a description of each.

In most cases, the number of times a symbol is repeated determines how that data is presented. Text information is displayed in an abbreviated form if the pattern letter is repeated less than four times. Otherwise, the unabbreviated form is used. For example, a zzzz pattern can display Pacific Daylight Time, and a zzz pattern can display PDT.

For numbers, the number of times a pattern letter is repeated determines how many digits are presented. For example, hh:mm:ss can present 01:51:15, but h:m:s displays the same time value as 1:51:15.

Finally, M or MM causes the month to be displayed as one or two digits. However, three or more repetitions of M cause the month to be displayed as a text string.

| Symbol | Description |
|---|---|
| a | AM or PM |
| d | Day of month (1–31) |
| h | Hour in AM/PM (1–12) |
| k | Hour in day (1–24) |
| m | Minute in hour (0–59) |
| s | Second in minute (0–59) |
| u | Day of week, with Monday being 1 |
| w | Week of year (1–52) |
| y | Year |
| z | Time zone |
| D | Day of year (1–366) |
| E | Day of week (for example, Thursday) |
| F | Day of week in month |
| G | Era (for example, AD or BC) |
| H | Hour in day (0–23) |
| K | Hour in AM/PM (0–11) |
| L | Month |
| M | Month |
| S | Millisecond in second |
| W | Week of month (1–5) |
| X | Time zone in ISO 8601 format |
| Y | Week year |
| Z | Time zone in RFC 822 format |

**Table 30-4** Formatting String Symbols for **SimpleDateFormat**

The following program shows how this class is used:

```
// Demonstrate SimpleDateFormat.
import java.text.*;
import java.util.*;

public class SimpleDateFormatDemo {
  public static void main(String args[]) {
    Date date = new Date();
    SimpleDateFormat sdf;
    sdf = new SimpleDateFormat("hh:mm:ss");
    System.out.println(sdf.format(date));
    sdf = new SimpleDateFormat("dd MMM yyyy hh:mm:ss zzz");
    System.out.println(sdf.format(date));
    sdf = new SimpleDateFormat("E MMM dd yyyy");
```

```
    System.out.println(sdf.format(date));
  }
}
```

Sample output from this program is shown here:

```
01:30:51
01 Jan 2014 01:30:51 CST
Wed Jan 01 2014
```



# The Time and Date API Added by JDK 8

In Chapter 19, Java's long-standing approach to handling date and time through the use of classes such as **Calendar** and **GregorianCalendar** was discussed. At the time of this writing, this traditional approach is still in widespread use and is something that all Java programmers need to be familiar with. However, with the release of JDK 8, Java now includes another approach to handling time and date. This new approach is defined in the following packages:

| Package | Description |
|---|---|
| java.time | Provides top-level classes that support time and date. |
| java.time.chrono | Supports alternative, non-Gregorian calendars. |
| java.time.format | Supports time and date formatting. |
| java.time.temporal | Supports extended date and time functionality. |
| java.time.zone | Supports time zones. |

These new packages define a large number of classes, interfaces, and enumerations that provide extensive, finely grained support for time and date operations. Because of the number of elements that comprise the new time and date API, it can seem fairly intimidating at first. However, it is well organized and logically structured. Its size reflects the detail of control and flexibility that it provides. Although it is far beyond the scope of this book to examine each element in this extensive API, we will look at several of its main classes. As you will see, these classes are sufficient for many uses.

## Time and Date Fundamentals

In **java.time** are defined several top-level classes that give you easy access to the time and date. Three of these are **LocalDate**, **LocalTime**, and **LocalDateTime**. As their names suggest, they encapsulate the local date, time, and date and time. Using these classes, it is easy to obtain the current date and time, format the date and time, and compare dates and times, among other operations.

**LocalDate** encapsulates a date that uses the default Gregorian calendar as specified by ISO 8601. **LocalTime** encapsulates a time, as specified by ISO 8601. **LocalDateTime** encapsulates both date and time. These classes contain a large number of methods that give you access to the date and time components, allow you to compare dates and times, add or subtract date or time components, and so on. Because a common naming convention for methods is employed, once you know how to use one of these classes, the others are easy to master.

**LocalDate**, **LocalTime**, and **LocalDateTime** do not define public constructors. Rather, to obtain an instance, you will use a factory method. One very convenient method is **now( )**, which is defined for all three classes. It returns the current date and/or time of the system. Each class defines several versions, but we will use its simplest form. Here is the version we will use as defined by **LocalDate**:

```
static LocalDate now( )
```

The version for **LocalTime** is shown here:

```
static LocalTime now( )
```

The version for **LocalDateTime** is shown here:

```
static LocalDateTime now( )
```

As you can see, in each case, an appropriate object is returned. The object returned by **now( )** can be displayed in its default, human-readable form by use of a **println( )** statement, for example. However, it is also possible to take full control over the formatting of date and time.

The following program uses **LocalDate** and **LocalTime** to obtain the current date and time and then displays them. Notice how **now( )** is called to retrieve the current date and time.

```
// A simple example of LocalDate and LocalTime.
import java.time.*;

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

    LocalDate curDate = LocalDate.now();
    System.out.println(curDate);

    LocalTime curTime = LocalTime.now();
    System.out.println(curTime);
  }
}
```

Sample output is shown here:

```
2014-01-01
14:03:41.436
```

The output reflects the default format that is given to the date and time. (The next section shows how to specify a different format.)

Because the preceding program displays both the current date and the current time, it could have been more easily written using the **LocalDateTime** class. In this approach, only a single instance needs to be created and only a single call to **now( )** is required, as shown here:

```
LocalDateTime curDateTime = LocalDateTime.now();
System.out.println(curDateTime);
```

Using this approach, the default output includes both date and time. Here is a sample:

```
2014-01-01T14:04:56.799
```

One other point: from a **LocalDateTime** instance, it is possible to obtain a reference to the date or time component by using the **toLocalDate( )** and **toLocalTime( )** methods, shown here:

LocalDate toLocalDate( )

LocalTime toLocalTime( )

Each returns a reference to the indicated element.



## Formatting Date and Time

Although the default formats shown in the preceding examples will be adequate for some uses, often you will want to specify a different format. Fortunately, this is easy to do because **LocalDate**, **LocalTime**, and **LocalDateTime** all provide the **format( )** method, shown here:

String format(DateTimeFormatter *fmtr*)

Here, *fmtr* specifies the instance of **DateTimeFormatter** that will provide the format.

**DateTimeFormatter** is packaged in **java.time.format**. To obtain a **DateTimeFormatter** instance, you will typically use one of its factory methods. Three are shown here:

static DateTimeFormatter ofLocalizedDate(FormatStyle *fmtDate*)

static DateTimeFormatter ofLocalizedTime(FormatStyle *fmtTime*)

static DateTimeFormatter ofLocalizedDateTime(FormatStyle *fmtDate*,
FormatStyle *fmtTime*)

Of course, the type of **DateTimeFormatter** that you create will be based on the type of object it will be operating on. For example, if you want to format the date in a **LocalDate** instance, then use **ofLocalizedDate( )**. The specific format is specified by the **FormatStyle** parameter.

**FormatStyle** is an enumeration that is packaged in **java.time.format**. It defines the following constants:

FULL

LONG

MEDIUM

SHORT

These specify the level of detail that will be displayed. (Thus, this form of **DateTimeFormatter** works similarly to **java.text.DateFormat**, described earlier in this chapter.)