

CHAPTER

19

java.util Part 2: More Utility Classes

This chapter continues our discussion of **java.util** by examining those classes and interfaces that are not part of the Collections Framework. These include classes that tokenize strings, work with dates, compute random numbers, bundle resources, and observe events. Also covered are the **Formatter** and **Scanner** classes which make it easy to write and read formatted data, and the new **Optional** class, which makes it easier to handle situations in which a value may be absent. Finally, the subpackages of **java.util** are summarized at the end of this chapter. Of particular interest is **java.util.function**, which defines several standard functional interfaces.

StringTokenizer

The processing of text often consists of parsing a formatted input string. *Parsing* is the division of text into a set of discrete parts, or *tokens*, which in a certain sequence can convey a semantic meaning. The **StringTokenizer** class provides the first step in this parsing process, often called the *lexer* (lexical analyzer) or *scanner*. **StringTokenizer** implements the **Enumeration** interface. Therefore, given an input string, you can enumerate the individual tokens contained in it using **StringTokenizer**.

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

The **StringTokenizer** constructors are shown here:

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

In all versions, *str* is the string that will be tokenized. In the first version, the default delimiters are used. In the second and third versions, *delimiters* is a string that specifies the delimiters. In the third version, if *delimAsToken* is **true**, then the delimiters are also returned

as tokens when the string is parsed. Otherwise, the delimiters are not returned. Delimiters are not returned as tokens by the first two forms.

Once you have created a **StringTokenizer** object, the **nextToken()** method is used to extract consecutive tokens. The **hasMoreTokens()** method returns **true** while there are more tokens to be extracted. Since **StringTokenizer** implements **Enumeration**, the **hasMoreElements()** and **nextElement()** methods are also implemented, and they act the same as **hasMoreTokens()** and **nextToken()**, respectively. The **StringTokenizer** methods are shown in Table 19-1.

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

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

class STDemo {
    static String in = "title=Java: The Complete Reference;" +
        "author=Schildt;" +
        "publisher=McGraw-Hill;" +
        "copyright=2014";

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

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

The output from this program is shown here:

```
title  Java: The Complete Reference
author  Schildt
publisher  McGraw-Hill
copyright  2014
```

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

Table 19-1 The Methods Defined by **StringTokenizer**

BitSet

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

```
BitSet()
BitSet(int size)
```

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

BitSet defines the methods listed in Table 19-2.

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

Table 19-2 The Methods Defined by **BitSet**

Method	Description
int nextSetBit(int <i>startIndex</i>)	Returns the index of the next set bit (that is, the next true bit), starting from the index specified by <i>startIndex</i> . If no bit is set, -1 is returned.
void or(BitSet <i>bitSet</i>)	ORs the contents of the invoking BitSet object with that specified by <i>bitSet</i> . The result is placed into the invoking object.
int previousClearBit(int <i>startIndex</i>)	Returns the index of the next cleared bit (that is, the next false bit) at or prior to the index specified by <i>startIndex</i> . If no cleared bit is found, -1 is returned.
int previousSetBit(int <i>startIndex</i>)	Returns the index of the next set bit (that is, the next true bit) at or prior to the index specified by <i>startIndex</i> . If no set bit is found, -1 is returned.
void set(int <i>index</i>)	Sets the bit specified by <i>index</i> .
void set(int <i>index</i> , boolean <i>v</i>)	Sets the bit specified by <i>index</i> to the value passed in <i>v</i> . true sets the bit; false clears the bit.
void set(int <i>startIndex</i> , int <i>endIndex</i>)	Sets the bits from <i>startIndex</i> to <i>endIndex</i> -1.
void set(int <i>startIndex</i> , int <i>endIndex</i> , boolean <i>v</i>)	Sets the bits from <i>startIndex</i> to <i>endIndex</i> -1 to the value passed in <i>v</i> . true sets the bits; false clears the bits.
int size()	Returns the number of bits in the invoking BitSet object.
InputStream stream()	Returns a stream that contains the bit positions, from low to high, that have set bits. (Added by JDK 8.)
byte[] toByteArray()	Returns a byte array that contains the invoking BitSet object.
long[] toLongArray()	Returns a long array that contains the invoking BitSet object.
String toString()	Returns the string equivalent of the invoking BitSet object.
static BitSet valueOf(byte[] <i>v</i>)	Returns a BitSet that contains the bits in <i>v</i> .
static BitSet valueOf(ByteBuffer <i>v</i>)	Returns a BitSet that contains the bits in <i>v</i> .
static BitSet valueOf(long[] <i>v</i>)	Returns a BitSet that contains the bits in <i>v</i> .
static BitSet valueOf(LongBuffer <i>v</i>)	Returns a BitSet that contains the bits in <i>v</i> .
void xor(BitSet <i>bitSet</i>)	XORs the contents of the invoking BitSet object with that specified by <i>bitSet</i> . The result is placed into the invoking object.

Table 19-2 The Methods Defined by **BitSet** (*continued*)

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

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

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

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

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

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

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

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

The output from this program is shown here. When **toString()** converts a **BitSet** object to its string equivalent, each set bit is represented by its bit position. Cleared bits are not shown.

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

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

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

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

bits2 XOR bits1:
{}
```

Optional, OptionalDouble, OptionalInt, and OptionalLong

JDK 8 adds classes called **Optional**, **OptionalDouble**, **OptionalInt**, and **OptionalLong** that offer a way to handle situations in which a value may or may not be present. In the past, you would normally use the value **null** to indicate that no value is present. However, this can lead to null pointer exceptions if an attempt is made to dereference a null reference. As a result, frequent checks for a **null** value were necessary to avoid generating an exception. These classes provide a better way to handle such situations.

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

```
class Optional<T>
```

Here, **T** specifies the type of value stored. It is important to understand that an **Optional** instance can either contain a value of type **T** or be empty. In other words, an **Optional** object does not necessarily contain a value. **Optional** does not define any constructors, but it does define several methods that let you work with **Optional** objects. For example, you can determine if a value is present, obtain the value if it is present, obtain a default value when no value is present, and construct an **Optional** value. The **Optional** methods are shown in Table 19-3.

Method	Description
static <T> Optional<T> empty()	Returns an object for which isPresent() returns false .
boolean equals(Object <i>optional</i>)	Returns true if the invoking object equals <i>optional</i> . Otherwise, returns false .
Optional<T> filter(Predicate<? super T> <i>condition</i>)	Returns an Optional instance that contains the same value as the invoking object if that value satisfies <i>condition</i> . Otherwise, an empty object is returned.
U Optional<U> flatMap(Function<? super T, Optional<U>> <i>mapFunc</i>)	Applies the mapping function specified by <i>mapFunc</i> to the invoking object if that object contains a value and returns the result. Returns an empty object otherwise.
T get()	Returns the value in the invoking object. However, if no value is present, NoSuchElementException is thrown.
int hashCode()	Returns a hashcode for the invoking object.
void ifPresent(Consumer<? super T> <i>func</i>)	Calls <i>func</i> if a value is present in the invoking object, passing the object to <i>func</i> . If no value is present, no action occurs.
boolean isPresent()	Returns true if the invoking object contains a value. Returns false if no value is present.
U Optional<U> map(Function<? super T, ? extends U>> <i>mapFunc</i>)	Applies the mapping function specified by <i>mapFunc</i> to the invoking object if that object contains a value and returns the result. Returns an empty object otherwise.
static <T> Optional<T> of(T <i>val</i>)	Creates an Optional instance that contains <i>val</i> and returns the result. The value of <i>val</i> must not be null .

Table 19-3 The Methods Defined by **Optional**

Method	Description
static <T> Optional<T> ofNullable(T val)	Creates an Optional instance that contains <i>val</i> and returns the result. However, if <i>val</i> is null , then an empty Optional instance is returned.
T orElse(T defVal)	If the invoking object contains a value, the value is returned. Otherwise, the value specified by <i>defVal</i> is returned.
T orElseGet(Supplier<? extends T> getFunc)	If the invoking object contains a value, the value is returned. Otherwise, the value obtained from <i>getFunc</i> is returned.
<X extends Throwable> T orElseThrow(Supplier<? extends X> excFunc) throws X extends Throwable	Returns the value in the invoking object. However, if no value is present, the exception generated by <i>excFunc</i> is thrown.
String toString()	Returns a string corresponding to the invoking object.

Table 19-3 The Methods Defined by **Optional** (continued)

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

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

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

The following program demonstrates these methods:

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

import java.util.*;

class OptionalDemo {
    public static void main(String args[]) {
        Optional<String> noVal = Optional.empty();
        Optional<String> hasVal = Optional.of("ABCDEFG");
    }
}
```

```

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

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

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

```

The output is shown here:

```

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

```

As the output shows, a value can be obtained from an **Optional** object only if one is present. This basic mechanism enables **Optional** to prevent null pointer exceptions.

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

Date

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

Date supports the following non-deprecated constructors:

```

Date()
Date(long millisec)

```

The first constructor initializes the object with the current date and time. The second constructor accepts one argument that equals the number of milliseconds that have elapsed since midnight, January 1, 1970. The nondeprecated methods defined by **Date** are shown in Table 19-4. **Date** also implements the **Comparable** interface.

Method	Description
boolean after(Date <i>date</i>)	Returns true if the invoking Date object contains a date that is later than the one specified by <i>date</i> . Otherwise, it returns false .
boolean before(Date <i>date</i>)	Returns true if the invoking Date object contains a date that is earlier than the one specified by <i>date</i> . Otherwise, it returns false .
Object clone()	Duplicates the invoking Date object.
int compareTo(Date <i>date</i>)	Compares the value of the invoking object with that of <i>date</i> . Returns 0 if the values are equal. Returns a negative value if the invoking object is earlier than <i>date</i> . Returns a positive value if the invoking object is later than <i>date</i> .
boolean equals(Object <i>date</i>)	Returns true if the invoking Date object contains the same time and date as the one specified by <i>date</i> . Otherwise, it returns false .
static Date from(Instant <i>t</i>)	Returns a Date object corresponding to the Instant object passed in <i>t</i> . (Added by JDK 8.)
long getTime()	Returns the number of milliseconds that have elapsed since January 1, 1970.
int hashCode()	Returns a hash code for the invoking object.
void setTime(long <i>time</i>)	Sets the time and date as specified by <i>time</i> , which represents an elapsed time in milliseconds from midnight, January 1, 1970.
Instant toInstant()	Returns an Instant object corresponding to the invoking Date object. (Added by JDK 8.)
String toString()	Converts the invoking Date object into a string and returns the result.

Table 19-4 The Nondeprecated Methods Defined by **Date**

As you can see by examining Table 19-4, the non-deprecated **Date** features do not allow you to obtain the individual components of the date or time. As the following program demonstrates, you can only obtain the date and time in terms of milliseconds, in its default string representation as returned by **toString()**, or (beginning with JDK 8) as an **Instant** object. To obtain more-detailed information about the date and time, you will use the **Calendar** class.

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

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

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

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

Sample output is shown here:

```
Wed Jan 01 11:11:44 CST 2014
Milliseconds since Jan. 1, 1970 GMT = 1388596304803
```

Calendar

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

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

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

A sampling of methods defined by **Calendar** are shown in Table 19-5.

Method	Description
abstract void add(int <i>which</i> , int <i>val</i>)	Adds <i>val</i> to the time or date component specified by <i>which</i> . To subtract, add a negative value. <i>which</i> must be one of the fields defined by Calendar , such as Calendar.HOUR .
boolean after(Object <i>calendarObj</i>)	Returns true if the invoking Calendar object contains a date that is later than the one specified by <i>calendarObj</i> . Otherwise, it returns false .
boolean before(Object <i>calendarObj</i>)	Returns true if the invoking Calendar object contains a date that is earlier than the one specified by <i>calendarObj</i> . Otherwise, it returns false .
final void clear()	Zeros all time components in the invoking object.
final void clear(int <i>which</i>)	Zeros the time component specified by <i>which</i> in the invoking object.
Object clone()	Returns a duplicate of the invoking object.
boolean equals(Object <i>calendarObj</i>)	Returns true if the invoking Calendar object contains a date that is equal to the one specified by <i>calendarObj</i> . Otherwise, it returns false .

Table 19-5 A Sampling of the Methods Defined by **Calendar**

Method	Description
int get(int <i>calendarField</i>)	Returns the value of one component of the invoking object. The component is indicated by <i>calendarField</i> . Examples of the components that can be requested are Calendar.YEAR , Calendar.MONTH , Calendar.MINUTE , and so forth.
static Locale[] getAvailableLocales()	Returns an array of Locale objects that contains the locales for which calendars are available.
static Calendar getInstance()	Returns a Calendar object for the default locale and time zone.
static Calendar getInstance(TimeZone <i>tz</i>)	Returns a Calendar object for the time zone specified by <i>tz</i> . The default locale is used.
static Calendar getInstance(Locale <i>locale</i>)	Returns a Calendar object for the locale specified by <i>locale</i> . The default time zone is used.
static Calendar getInstance(TimeZone <i>tz</i> , Locale <i>locale</i>)	Returns a Calendar object for the time zone specified by <i>tz</i> and the locale specified by <i>locale</i> .
final Date getTime()	Returns a Date object equivalent to the time of the invoking object.
TimeZone getTimeZone()	Returns the time zone for the invoking object.
final boolean isSet(int <i>which</i>)	Returns true if the specified time component is set. Otherwise, it returns false .
void set(int <i>which</i> , int <i>val</i>)	Sets the date or time component specified by <i>which</i> to the value specified by <i>val</i> in the invoking object. <i>which</i> must be one of the fields defined by Calendar , such as Calendar.HOUR .
final void set(int <i>year</i> , int <i>month</i> , int <i>dayOfMonth</i>)	Sets various date and time components of the invoking object.
final void set(int <i>year</i> , int <i>month</i> , int <i>dayOfMonth</i> , int <i>hours</i> , int <i>minutes</i>)	Sets various date and time components of the invoking object.
final void set(int <i>year</i> , int <i>month</i> , int <i>dayOfMonth</i> , int <i>hours</i> , int <i>minutes</i> , int <i>seconds</i>)	Sets various date and time components of the invoking object.
final void setTime(Date <i>d</i>)	Sets various date and time components of the invoking object. This information is obtained from the Date object <i>d</i> .
void setTimeZone(TimeZone <i>tz</i>)	Sets the time zone for the invoking object to that specified by <i>tz</i> .
final Instant toInstant()	Returns an Instant object corresponding to the invoking Calendar instance. (Added by JDK 8.)

Table 19-5 A Sampling of the Methods Defined by **Calendar** (*continued*)

Calendar defines the following **int** constants, which are used when you get or set components of the calendar. (The ones with the suffix **FORMAT** or **STANDALONE** were added by JDK 8.)

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

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

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

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

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

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

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

    }
}
```

```
System.out.print(calendar.get(Calendar.MINUTE) + ":");

System.out.println(calendar.get(Calendar.SECOND));

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

}
```

Sample output is shown here:

```
Date: Jan 1 2014
Time: 11:29:39
Updated time: 10:29:22
```

GregorianCalendar

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

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

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

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

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

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

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

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

```
boolean isLeapYear(int year)
```

This method returns **true** if *year* is a leap year and **false** otherwise. JDK 8 also adds the following methods: **from()** and **toZonedDateTime()**, which support the new date and time API, and **getCalendarType()**, which returns the calendar type as a string, which is “gregory”.

The following program demonstrates **GregorianCalendar**:

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

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

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

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

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

        // Test if the current year is a leap year
        if(gcalendar.isLeapYear(year)) {
            System.out.println("The current year is a leap year");
        }
        else {
            System.out.println("The current year is not a leap year");
        }
    }
}
```

Sample output is shown here:

```
Date: Jan 1 2014
Time: 1:45:5
The current year is not a leap year
```

TimeZone

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

A sampling of methods defined by **TimeZone** is given in Table 19-6.

Method	Description
<code>Object clone()</code>	Returns a TimeZone -specific version of <code>clone()</code> .
<code>static String[] getAvailableIDs()</code>	Returns an array of String objects representing the names of all time zones.
<code>static String[] getAvailableIDs(int timeDelta)</code>	Returns an array of String objects representing the names of all time zones that are <i>timeDelta</i> offset from GMT.
<code>static TimeZone getDefault()</code>	Returns a TimeZone object that represents the default time zone used on the host computer.
<code>StringgetID()</code>	Returns the name of the invoking TimeZone object.
<code>abstract int getOffset(int era, int year, int month, int dayOfMonth, int dayOfWeek, int millis)</code>	Returns the offset that should be added to GMT to compute local time. This value is adjusted for daylight saving time. The parameters to the method represent date and time components.
<code>abstract int getRawOffset()</code>	Returns the raw offset (in milliseconds) that should be added to GMT to compute local time. This value is not adjusted for daylight saving time.
<code>static TimeZone getTimeZone(String tzName)</code>	Returns the TimeZone object for the time zone named <i>tzName</i> .
<code>abstract boolean inDaylightTime(Date d)</code>	Returns true if the date represented by <i>d</i> is in daylight saving time in the invoking object. Otherwise, it returns false .
<code>static void setDefault(TimeZone tz)</code>	Sets the default time zone to be used on this host. <i>tz</i> is a reference to the TimeZone object to be used.
<code>void setID(String tzName)</code>	Sets the name of the time zone (that is, its ID) to that specified by <i>tzName</i> .
<code>abstract void setRawOffset(int millis)</code>	Sets the offset in milliseconds from GMT.
<code>ZoneId toZoneId()</code>	Converts the invoking object into a ZoneId and returns the result. ZoneId is packaged in java.time . (Added by JDK 8.)
<code>abstract boolean useDaylightTime()</code>	Returns true if the invoking object uses daylight saving time. Otherwise, it returns false .

Table 19-6 A Sampling of the Methods Defined by **TimeZone**

SimpleTimeZone

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

SimpleTimeZone defines four constructors. One is

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

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

The second **SimpleTimeZone** constructor is

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

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

The third **SimpleTimeZone** constructor is

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

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

The fourth **SimpleTimeZone** constructor is:

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

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

STANDARD_TIME	WALL_TIME	UTC_TIME
---------------	-----------	----------

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

Locale

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

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

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

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

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

The constructors for **Locale** are

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

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

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

```
static void setDefault(Locale localeObj)
```

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

Some other interesting methods are the following:

```
final String getDisplayCountry()
final String getDisplayLanguage()
final String getDisplayName()
```

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

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

```
static Locale getDefault()
```

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

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

Random

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

```
Random()
Random(long seed)
```

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

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

The core public methods defined by **Random** are shown in Table 19-7. These are the methods that have been available in **Random** for several years (many since Java 1.0) and are widely used.

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

Here is an example that demonstrates the sequence produced by **nextGaussian()**. It obtains 100 random Gaussian values and averages these values. The program also counts the

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

Table 19-7 The Core Methods Defined by **Random**

number of values that fall within two standard deviations, plus or minus, using increments of 0.5 for each category. The result is graphically displayed sideways on the screen.

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

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

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

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

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

JDK 8 adds three new methods to `Random` that support the new stream API (see Chapter 29). They are called `doubles()`, `ints()`, and `longs()`, and each returns a reference

to a stream that contains a sequence of pseudorandom values of the specified type. Each method defines several overloads. Here are their simplest forms:

`DoubleStream doubles()`

`IntStream ints()`

`LongStream longs()`

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

Observable

The **Observable** class is used to create subclasses that other parts of your program can observe. When an object of such a subclass undergoes a change, observing classes are notified. Observing classes must implement the **Observer** interface, which defines the **update()** method. The **update()** method is called when an observer is notified of a change in an observed object.

Observable defines the methods shown in Table 19-8. An object that is being observed must follow two simple rules. First, if it has changed, it must call **setChanged()**. Second, when it is ready to notify observers of this change, it must call **notifyObservers()**. This causes the **update()** method in the observing object(s) to be called. Be careful—if the

Method	Description
<code>void addObserver(Observer obj)</code>	Adds <i>obj</i> to the list of objects observing the invoking object.
<code>protected void clearChanged()</code>	Calling this method returns the status of the invoking object to "unchanged."
<code>int countObservers()</code>	Returns the number of objects observing the invoking object.
<code>void deleteObserver(Observer obj)</code>	Removes <i>obj</i> from the list of objects observing the invoking object.
<code>void deleteObservers()</code>	Removes all observers for the invoking object.
<code>boolean hasChanged()</code>	Returns true if the invoking object has been modified and false if it has not.
<code>void notifyObservers()</code>	Notifies all observers of the invoking object that it has changed by calling update() . A null is passed as the second argument to update() .
<code>void notifyObservers(Object obj)</code>	Notifies all observers of the invoking object that it has changed by calling update() . <i>obj</i> is passed as an argument to update() .
<code>protected void setChanged()</code>	Called when the invoking object has changed.

Table 19-8 The Methods Defined by **Observable**

object calls **notifyObservers()** without having previously called **setChanged()**, no action will take place. The observed object must call both **setChanged()** and **notifyObservers()** before **update()** will be called.

Notice that **notifyObservers()** has two forms: one that takes an argument and one that does not. If you call **notifyObservers()** with an argument, this object is passed to the observer's **update()** method as its second parameter. Otherwise, **null** is passed to **update()**. You can use the second parameter for passing any type of object that is appropriate for your application.

The Observer Interface

To observe an observable object, you must implement the **Observer** interface. This interface defines only the one method shown here:

```
void update(Observable observOb, Object arg)
```

Here, *observOb* is the object being observed, and *arg* is the value passed by **notifyObservers()**. The **update()** method is called when a change in the observed object takes place.

An Observer Example

Here is an example that demonstrates an observable object. It creates an observer class, called **Watcher**, that implements the **Observer** interface. The class being monitored is called **BeingWatched**. It extends **Observable**. Inside **BeingWatched** is the method **counter()**, which simply counts down from a specified value. It uses **sleep()** to wait a tenth of a second between counts. Each time the count changes, **notifyObservers()** is called with the current count passed as its argument. This causes the **update()** method inside **Watcher** to be called, which displays the current count. Inside **main()**, a **Watcher** and a **BeingWatched** object, called **observing** and **observed**, respectively, are created. Then, **observing** is added to the list of observers for **observed**. This means that **observing.update()** will be called each time **counter()** calls **notifyObservers()**.

```
/* Demonstrate the Observable class and the
   Observer interface.
*/
import java.util.*;

// This is the observing class.
class Watcher implements Observer {
    public void update(Observable obj, Object arg) {
        System.out.println("update() called, count is " +
                           ((Integer)arg).intValue());
    }
}

// This is the class being observed.
class BeingWatched extends Observable {
    void counter(int period) {
        for( ; period >=0; period--) {
            setChanged();
            notifyObservers(new Integer(period));
        }
    }
}
```

```
        try {
            Thread.sleep(100);
        } catch(InterruptedException e) {
            System.out.println("Sleep interrupted");
        }
    }
}

class ObserverDemo {
    public static void main(String args[]) {
        BeingWatched observed = new BeingWatched();
        Watcher observing = new Watcher();

        /* Add the observing to the list of observers for
           observed object. */
        observed.addObserver(observing);

        observed.counter(10);
    }
}
```

The output from this program is shown here:

```
update() called, count is 10
update() called, count is 9
update() called, count is 8
update() called, count is 7
update() called, count is 6
update() called, count is 5
update() called, count is 4
update() called, count is 3
update() called, count is 2
update() called, count is 1
update() called, count is 0
```

More than one object can be an observer. For example, the following program implements two observing classes and adds an object of each class to the **BeingWatched** observer list. The second observer waits until the count reaches zero and then rings the bell.

```
        }
```

```
// This is the second observing class.
class Watcher2 implements Observer {
    public void update(Observable obj, Object arg) {
        // Ring bell when done
        if(((Integer)arg).intValue() == 0)
            System.out.println("Done" + '\7');
    }
}

// This is the class being observed.
class BeingWatched extends Observable {
    void counter(int period) {
        for( ; period >=0; period--) {
            setChanged();
            notifyObservers(new Integer(period));
            try {
                Thread.sleep(100);
            } catch(InterruptedException e) {
                System.out.println("Sleep interrupted");
            }
        }
    }
}

class TwoObservers {
    public static void main(String args[]) {
        BeingWatched observed = new BeingWatched();
        Watcher1 observing1 = new Watcher1();
        Watcher2 observing2 = new Watcher2();

        // add both observers
        observed.addObserver(observing1);
        observed.addObserver(observing2);

        observed.counter(10);
    }
}
```

The **Observable** class and the **Observer** interface allow you to implement sophisticated program architectures based on the document/view methodology.

Timer and TimerTask

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

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

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

```
protected TimerTask()
```

TimerTask defines the methods shown in Table 19-9. Notice that `run()` is abstract, which means that it must be overridden. The `run()` method, defined by the **Runnable** interface, contains the code that will be executed. Thus, the easiest way to create a timer task is to extend **TimerTask** and override `run()`.

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

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

The first version creates a **Timer** object that runs as a normal thread. The second uses a daemon thread if `DThread` is `true`. A daemon thread will execute only as long as the rest of the program continues to execute. The third and fourth constructors allow you to specify a name for the **Timer** thread. The methods defined by **Timer** are shown in Table 19-9.

Once a **Timer** has been created, you will schedule a task by calling `schedule()` on the **Timer** that you created. As Table 19-10 shows, there are several forms of `schedule()` which allow you to schedule tasks in a variety of ways.

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

Table 19-9 The Methods Defined by **TimerTask**

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

Table 19-10 The Methods Defined by **Timer**

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

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

```
// Demonstrate Timer and TimerTask.

import java.util.*;

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

class TTest {
```

```

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

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

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

    myTimer.cancel();
}
}

```

Currency

The **Currency** class encapsulates information about a currency. It defines no constructors. The methods supported by **Currency** are shown in Table 19-11. The following program demonstrates **Currency**:

```

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

```

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

Table 19-11 The Methods Defined by **Currency**

```
class CurDemo {  
    public static void main(String args[]) {  
        Currency c;  
  
        c = Currency.getInstance(Locale.US);  
  
        System.out.println("Symbol: " + c.getSymbol());  
        System.out.println("Default fractional digits: " +  
                           c.getDefaultFractionDigits());  
    }  
}
```

The output is shown here:

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

Formatter

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

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

The Formatter Constructors

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

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

```
Formatter()  
Formatter(Appendable buf)  
Formatter(Appendable buf, Locale loc)  
Formatter(String filename)  
      throws FileNotFoundException  
Formatter(String filename, String charset)  
      throws FileNotFoundException, UnsupportedEncodingException
```

```

Formatter(File outF)
throws FileNotFoundException
Formatter(OutputStream outStrm)

```

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

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

The Formatter Methods

Formatter defines the methods shown in Table 19-12.

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

Table 19-12 The Methods Defined by **Formatter**

Formatting Basics

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

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

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

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

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

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

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

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

The **format()** method accepts a wide variety of format specifiers, which are shown in Table 19-13. Notice that many specifiers have both upper- and lowercase forms. When an uppercase specifier is used, then letters are shown in uppercase. Otherwise, the upper- and

Format Specifier	Conversion Applied
%a %A	Floating-point hexadecimal
%b %B	Boolean
%c	Character
%d	Decimal integer
%h %H	Hash code of the argument
%e %E	Scientific notation
%f	Decimal floating-point

Table 19-13 The Format Specifiers

Format Specifier	Conversion Applied
%g %G	Uses %e or %f, based on the value being formatted and the precision
%o	Octal integer
%n	Inserts a newline character
%s %S	String
%t %T	Time and date
%x %X	Integer hexadecimal
%%	Inserts a % sign

Table 19-13 The Format Specifiers (*continued*)

lowercase specifiers perform the same conversion. It is important to understand that Java type-checks each format specifier against its corresponding argument. If the argument doesn't match, an **IllegalFormatException** is thrown.

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

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

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

Here is a short program that puts together all of the pieces, showing how to create and display a formatted string:

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

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

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

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

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

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

Formatting Strings and Characters

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

Formatting Numbers

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

`x.ddddde+/-yy`

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

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

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

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

It produces the following output:

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

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

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

produces this output:

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

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

`0x1.sigpexp`

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

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

produces this output:

```
0x1.0p9
```

Formatting Time and Date

One of the more powerful conversion specifiers is **%t**. It lets you format time and date information. The **%t** specifier works a bit differently than the others because it requires the use of a suffix to describe the portion and precise format of the time or date desired. The suffixes are shown in Table 19-14. For example, to display minutes, you would use **%tM**, where **M** indicates minutes in a two-character field. The argument corresponding to the **%t** specifier must be of type **Calendar**, **Date**, **Long**, or **long**.

Here is a program that demonstrates several of the formats:

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

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

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

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

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

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

Suffix	Replaced By
a	Abbreviated weekday name
A	Full weekday name
b	Abbreviated month name
B	Full month name
c	Standard date and time string formatted as <i>day month date hh:mm:ss tzzone year</i>
C	First two digits of year
d	Day of month as a decimal (01—31)
D	month/day/year
e	Day of month as a decimal (1—31)
F	year-month-day
h	Abbreviated month name
H	Hour (00 to 23)
I	Hour (01 to 12)
j	Day of year as a decimal (001 to 366)
k	Hour (0 to 23)
l	Hour (1 to 12)
L	Millisecond (000 to 999)
m	Month as decimal (01 to 13)
M	Minute as decimal (00 to 59)
N	Nanosecond (000000000 to 999999999)
p	Locale's equivalent of AM or PM in lowercase
Q	Milliseconds from 1/1/1970
r	<i>hh:mm:ss</i> (12-hour format)
R	<i>hh:mm</i> (24-hour format)
S	Seconds (00 to 60)
s	Seconds from 1/1/1970 UTC
T	<i>hh:mm:ss</i> (24-hour format)
y	Year in decimal without century (00 to 99)
Y	Year in decimal including century (0001 to 9999)
z	Offset from UTC
Z	Time zone name

Table 19-14 The Time and Date Format Suffixes

Sample output is shown here:

```
03:15:34 PM
Wed Jan 01 15:15:34 CST 2014
3:15
January Jan 01
```

The %n and %% Specifiers

The %n and %% format specifiers differ from the others in that they do not match an argument. Instead, they are simply escape sequences that insert a character into the output sequence. The %n inserts a newline. The %% inserts a percent sign. Neither of these characters can be entered directly into the format string. Of course, you can also use the standard escape sequence \n to embed a newline character.

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

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

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

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

It displays the following output:

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

Specifying a Minimum Field Width

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

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

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

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

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

This program produces the following output:

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

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

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

```
// Create a table of squares and cubes.  
import java.util.*;  
  
class FieldWidthDemo {  
    public static void main(String args[]) {  
        Formatter fmt;  
  
        for(int i=1; i <= 10; i++) {  
            fmt = new Formatter();  
            fmt.format("%4d %4d %4d", i, i*i, i*i*i);  
            System.out.println(fmt);  
            fmt.close();  
        }  
    }  
}
```

Its output is shown here:

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

Specifying Precision

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

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

Applied to strings, the precision specifier specifies the maximum field length. For example, `%5.7s` displays a string of at least five and not exceeding seven characters long. If the string is longer than the maximum field width, the end characters will be truncated.

The following program illustrates the precision specifier:

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

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

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

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

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

It produces the following output:

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

Using the Format Flags

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

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

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

Justifying Output

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

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

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

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

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

It produces the following output:

123.12	
123.12	

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

The Space, +, 0, and (Flags

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

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

creates this string:

```
+100
```

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

```
// Demonstrate the space format specifiers.  
import java.util.*;  
  
class FormatDemo5 {  
    public static void main(String args[]) {  
        Formatter fmt = new Formatter();  
  
        fmt.format ("% d", -100);  
        System.out.println(fmt);  
        fmt.close();  
  
        fmt = new Formatter();  
        fmt.format ("% d", 100);  
        System.out.println(fmt);  
        fmt.close();  
  
        fmt = new Formatter();  
        fmt.format ("% d", -200);  
        System.out.println(fmt);  
        fmt.close();  
    }  
}
```

The output is shown here:

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

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

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

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

creates this string:

```
(100)
```

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

The Comma Flag

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

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

creates this string:

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

The # Flag

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

The Uppercase Option

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

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

For example, this call:

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

creates this string:

FA

This call:

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

creates this string:

1.231234E+02

Using an Argument Index

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

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

n\$

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

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

It produces this string:

30 10 20

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

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

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

It produces the following string:

255 in hex is ff

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

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

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

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

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

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

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

Here is sample output:

```
Today is day 1 of January, 2014
```

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

Closing a Formatter

In general, you should close a **Formatter** when you are done using it. Doing so frees any resources that it was using. This is especially important when formatting to a file, but it can be important in other cases, too. As the previous examples have shown, one way to close a **Formatter** is to explicitly call **close()**. However, beginning with JDK 7, **Formatter** implements the **AutoCloseable** interface. This means that it supports the **try-with-resources** statement. Using this approach, the **Formatter** is automatically closed when it is no longer needed.

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

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

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

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

```

        }
    }
}
```

The output is the same as before.

The Java printf() Connection

Although there is nothing technically wrong with using **Formatter** directly (as the preceding examples have done) when creating output that will be displayed on the console, there is a more convenient alternative: the **printf()** method. The **printf()** method automatically uses **Formatter** to create a formatted string. It then displays that string on **System.out**, which is the console by default. The **printf()** method is defined by both **PrintStream** and **PrintWriter**. The **printf()** method is described in Chapter 20.

Scanner

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

The Scanner Constructors

Scanner defines the constructors shown in Table 19-15. In general, a **Scanner** can be created for a **String**, an **InputStream**, a **File**, or any object that implements the **Readable** or **ReadableByteChannel** interfaces. Here are some examples.

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

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

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

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

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

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

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

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

Scanning Basics

Once you have created a **Scanner**, it is a simple matter to use it to read formatted input. In general, a **Scanner** reads *tokens* from the underlying source that you specified when the **Scanner** was created. As it relates to **Scanner**, a token is a portion of input that is delineated

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

Table 19-15 The **Scanner** Constructors

by a set of delimiters, which is whitespace by default. A token is read by matching it with a particular *regular expression*, which defines the format of the data. Although **Scanner** allows you to define the specific type of expression that its next input operation will match, it includes many predefined patterns, which match the primitive types, such as **int** and **double**, and strings. Thus, often you won't need to specify a pattern to match.

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

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

As the preceding indicates, **Scanner** defines two sets of methods that enable you to read input. The first are the **hasNextX** methods, which are shown in Table 19-16. These methods determine if the specified type of input is available. For example, calling **hasnextInt()** returns **true** only if the next token to be read is an integer. If the desired data is available, then you read it by calling one of **Scanner**'s **nextX** methods, which are shown in Table 19-17.

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

Table 19-16 The **Scanner** **hasNext** Methods

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

Table 19-17 The **Scanner next** Methods

For example, to read the next integer, call **nextInt()**. The following sequence shows how to read a list of integers from the keyboard.

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

// Read a list of integers.
while(conin.hasNextInt()) {
```

```
i = conin.nextInt();
// ...
}
```

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

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

Some Scanner Examples

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

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

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

        int count = 0;
        double sum = 0.0;

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

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

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

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

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

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

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

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

```
// Use Scanner to compute an average of the values in a file.  
import java.util.*;  
import java.io.*;  
  
class AvgFile {  
    public static void main(String args[])  
        throws IOException {  
  
        int count = 0;  
        double sum = 0.0;  
  
        // Write output to a file.  
        FileWriter fout = new FileWriter("test.txt");  
        fout.write("2 3.4 5 6 7.4 9.1 10.5 done");  
        fout.close();  
  
        FileReader fin = new FileReader("Test.txt");  
  
        Scanner src = new Scanner(fin);  
  
        // Read and sum numbers.  
        while(src.hasNext()) {  
            if(src.hasNextDouble()) {  
                sum += src.nextDouble();  
                count++;  
            }  
            else {  
                // ignore non-numbers  
            }  
        }  
        System.out.println("Average is " + sum / count);  
    }  
}
```

```

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

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

```

Here is the output:

```
Average is 6.2
```

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

Beginning with JDK 7, **Scanner** also implements the **AutoCloseable** interface. This means that it can be managed by a **try-with-resources** block. As explained in Chapter 13, when **try-with-resources** is used, the scanner is automatically closed when the block ends. For example, **src** in the preceding program could have been managed like this:

```

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

```

To clearly demonstrate the closing of a **Scanner**, the following examples will call **close()** explicitly. (Doing so also allows them to be compiled by versions of Java prior to JDK 7.) However, the **try-with-resources** approach is more streamlined and can help prevent errors. Its use is recommended for new code.

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

You can use `Scanner` to read input that contains several different types of data—even if the order of that data is unknown in advance. You must simply check what type of data is available before reading it. For example, consider this program:

```
// Use Scanner to read various types of data from a file.  
import java.util.*;  
import java.io.*;  
  
class ScanMixed {  
    public static void main(String args[])  
        throws IOException {  
  
        int i;  
        double d;  
        boolean b;  
        String str;  
  
        // Write output to a file.  
        FileWriter fout = new FileWriter("test.txt");  
        fout.write("Testing Scanner 10 12.2 one true two false");  
        fout.close();  
  
        FileReader fin = new FileReader("Test.txt");  
  
        Scanner src = new Scanner(fin);  
  
        // Read to end.  
        while(src.hasNext()) {  
            if(src.hasNextInt()) {  
                i = src.nextInt();  
                System.out.println("int: " + i);  
            }  
            else if(src.hasNextDouble()) {  
                d = src.nextDouble();  
                System.out.println("double: " + d);  
            }  
            else if(src.hasNextBoolean()) {  
                b = src.nextBoolean();  
                System.out.println("boolean: " + b);  
            }  
            else {  
                str = src.next();  
                System.out.println("String: " + str);  
            }  
        }  
  
        src.close();  
    }  
}
```

Here is the output:

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

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

Setting Delimiters

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

```
Scanner useDelimiter(String pattern)
Scanner useDelimiter(Pattern pattern)
```

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

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

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

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

        int count = 0;
        double sum = 0.0;

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

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

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

        Scanner src = new Scanner(fin);
```

```

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

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

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

```

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

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

Pattern delimiter()

Other Scanner Features

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

```

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

```

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

```

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

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

```

```

Scanner conin = new Scanner(instr);

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

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

conin.close();
}
}

```

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

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

`String findWithinHorizon(Pattern pattern, int count)`

`String findWithinHorizon(String pattern, int count)`

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

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

`Scanner skip(Pattern pattern)`

`Scanner skip(String pattern)`

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

Other **Scanner** methods include **radix()**, which returns the default radix used by the **Scanner**; **useRadix()**, which sets the radix; **reset()**, which resets the scanner; and **close()**, which closes the scanner.

The ResourceBundle, ListResourceBundle, and PropertyResourceBundle Classes

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

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

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

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

The methods defined by **ResourceBundle** are summarized in Table 19-18. One important point: **null** keys are not allowed and several of the methods will throw a **NullPointerException** if **null** is passed as the key. Notice the nested class **ResourceBundle.Control**. It is used to control the resource-bundle loading process.

Method	Description
static final void clearCache()	Deletes all resource bundles from the cache that were loaded by the default class loader.
static final void clearCache(ClassLoader <i>ldr</i>)	Deletes all resource bundles from the cache that were loaded by <i>ldr</i> .
boolean containsKey(String <i>k</i>)	Returns true if <i>k</i> is a key within the invoking resource bundle (or its parent).
String getBaseBundleName()	Returns the resource bundle's base name if available. Returns null otherwise. (Added by JDK 8.)
static final ResourceBundle getBundle(String <i>familyName</i>)	Loads the resource bundle with a family name of <i>familyName</i> using the default locale and the default class loader. Throws MissingResourceException if no resource bundle matching the family name specified by <i>familyName</i> is available.
static final ResourceBundle getBundle(String <i>familyName</i> , Locale <i>loc</i>)	Loads the resource bundle with a family name of <i>familyName</i> using the specified locale and the default class loader. Throws MissingResourceException if no resource bundle matching the family name specified by <i>familyName</i> is available.
static ResourceBundle getBundle(String <i>familyName</i> , Locale <i>loc</i> , ClassLoader <i>ldr</i>)	Loads the resource bundle with a family name of <i>familyName</i> using the specified locale and the specified class loader. Throws MissingResourceException if no resource bundle matching the family name specified by <i>familyName</i> is available.

Table 19-18 The Methods Defined by **ResourceBundle**

Method	Description
static final ResourceBundle getBundle(String <i>familyName</i> , ResourceBundle.Control <i>cntl</i>)	Loads the resource bundle with a family name of <i>familyName</i> using the default locale and the default class loader. The loading process is under the control of <i>cntl</i> . Throws MissingResourceException if no resource bundle matching the family name specified by <i>familyName</i> is available.
static final ResourceBundle getBundle(String <i>familyName</i> , Locale <i>loc</i> , ResourceBundle.Control <i>cntl</i>)	Loads the resource bundle with a family name of <i>familyName</i> using the specified locale and the default class loader. The loading process is under the control of <i>cntl</i> . Throws MissingResourceException if no resource bundle matching the family name specified by <i>familyName</i> is available.
static ResourceBundle getBundle(String <i>familyName</i> , Locale <i>loc</i> , ClassLoader <i>ldr</i> , ResourceBundle.Control <i>cntl</i>)	Loads the resource bundle with a family name of <i>familyName</i> using the specified locale and the specified class loader. The loading process is under the control of <i>cntl</i> . Throws MissingResourceException if no resource bundle matching the family name specified by <i>familyName</i> is available.
abstract Enumeration<String> getKeys()	Returns the resource bundle keys as an enumeration of strings. Any parent's keys are also obtained.
Locale getLocale()	Returns the locale supported by the resource bundle.
final Object getObject(String <i>k</i>)	Returns the object associated with the key passed via <i>k</i> . Throws MissingResourceException if <i>k</i> is not in the resource bundle.
final String getString(String <i>k</i>)	Returns the string associated with the key passed via <i>k</i> . Throws MissingResourceException if <i>k</i> is not in the resource bundle. Throws ClassCastException if the object associated with <i>k</i> is not a string.
final String[] getStringArray(String <i>k</i>)	Returns the string array associated with the key passed via <i>k</i> . Throws MissingResourceException if <i>k</i> is not in the resource bundle. Throws MissingResourceException if the object associated with <i>k</i> is not a string array.
protected abstract Object handleGetObject(String <i>k</i>)	Returns the object associated with the key passed via <i>k</i> . Returns null if <i>k</i> is not in the resource bundle.
protected Set<String> handleKeySet()	Returns the resource bundle keys as a set of strings. No parent's keys are obtained. Also, keys with null values are not obtained.
Set<String> keySet()	Returns the resource bundle keys as a set of strings. Any parent keys are also obtained.
protected void setParent(ResourceBundle <i>parent</i>)	Sets <i>parent</i> as the parent bundle for the resource bundle. When a key is looked up, the parent will be searched if the key is not found in the invoking resource object.

Table 19-18 The Methods Defined by **ResourceBundle** (continued)

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

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

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

Here is an example that demonstrates using a resource bundle. The resource bundle has the family name **SampleRB**. Two resource bundle classes of this family are created by extending **ListResourceBundle**. The first is called **SampleRB**, and it is the default bundle (which uses English). It is shown here:

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

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

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

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

        return resources;
    }
}
```

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

```
import java.util.*;

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

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

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

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

```

        return resources;
    }
}

```

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

```

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

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

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

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

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

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

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

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

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

```

The output from the program is shown here:

```

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

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

```

Miscellaneous Utility Classes and Interfaces

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

Base64	Supports Base64 encoding. Encoder and Decoder nested classes are also defined. (Added by JDK 8.)
DoubleSummaryStatistics	Supports the compilation of double values. The following statistics are available: average, minimum, maximum, count, and sum. (Added by JDK 8.)
EventListenerProxy	Extends the EventListener class to allow additional parameters. See Chapter 24 for a discussion of event listeners.
EventObject	The superclass for all event classes. Events are discussed in Chapter 24.
FormattableFlags	Defines formatting flags that are used with the Formattable interface.
IntSummaryStatistics	Supports the compilation of int values. The following statistics are available: average, minimum, maximum, count, and sum. (Added by JDK 8.)
Objects	Various methods that operate on objects.
PropertyPermission	Manages property permissions.
ServiceLoader	Provides a means of finding service providers.
StringJoiner	Supports the concatenation of CharSequences , which may include a separator, a prefix, and a suffix. (Added by JDK 8.)
UUID	Encapsulates and manages Universally Unique Identifiers (UUIDs).

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

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

The **java.util** Subpackages

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

- **java.util.concurrent**
- **java.util.concurrent.atomic**
- **java.util.concurrent.locks**
- **java.util.function**
- **java.util.jar**
- **java.util.logging**
- **java.util.prefs**

- `java.util.regex`
- `java.util.spi`
- `java.util.stream`
- `java.util.zip`

Each is briefly examined here.

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

The `java.util.concurrent` package along with its two subpackages, `java.util.concurrent.atomic` and `java.util.concurrent.locks`, support concurrent programming. These packages provide a high-performance alternative to using Java's built-in synchronization features when thread-safe operation is required. Beginning with JDK 7, `java.util.concurrent` also provides the Fork/Join Framework. These packages are examined in detail in Chapter 28.

java.util.function

The `java.util.function` package defines several predefined functional interfaces that you can use when creating lambda expressions or method references. They are also widely used throughout the Java API. The functional interfaces defined by `java.util.function` are shown in Table 19-19 along with a synopsis of their abstract methods. Be aware that some of these interfaces also define default or static methods that supply additional functionality. You will want to explore them fully on your own. (For a discussion of the use of functional interfaces, see Chapter 15.)

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

Table 19-19 Functional Interfaces Defined by `java.util.function` and Their Abstract Methods

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

Table 19-19 Functional Interfaces Defined by **java.util.function** and Their Abstract Methods (*continued*)

Interface	Abstract Method
IntUnaryOperator	int applyAsInt(int <i>val</i>) Description: Acts on an int and returns an int result.
LongBinaryOperator	long applyAsLong(long <i>val1</i> , long <i>val2</i>) Description: Acts on two long values and returns a long result.
LongConsumer	void accept(long <i>val</i>) Description: Acts on <i>val</i> .
LongFunction<R>	R apply(long <i>val</i>) Description: Acts on a long value and returns the result.
LongPredicate	boolean test(long <i>val</i>) Description: Returns true if <i>val</i> satisfies the condition defined by <i>test()</i> and false otherwise.
LongSupplier	long getAsLong() Description: Returns a long result.
LongToDoubleFunction	double applyAsDouble(long <i>val</i>) Description: Acts on a long value and returns the result as a double .
LongToIntFunction	int applyAsInt(long <i>val</i>) Description: Acts on a long value and returns the result as an int .
LongUnaryOperator	long applyAsLong(long <i>val</i>) Description: Acts on a long and returns a long result.
ObjDoubleConsumer<T>	void accept(T <i>val1</i> , double <i>val2</i>) Description: Acts on <i>val1</i> and the double value <i>val2</i> .
ObjIntConsumer<T>	void accept(T <i>val1</i> , int <i>val2</i>) Description: Acts on <i>val1</i> and the int value <i>val2</i> .
ObjLongConsumer<T>	void accept(T <i>val1</i> , long <i>val2</i>) Description: Acts on <i>val1</i> and the long value <i>val2</i> .
Predicate<T>	boolean test(T <i>val</i>) Description: Returns true if <i>val</i> satisfies the condition defined by <i>test()</i> and false otherwise.
Supplier<T>	T get() Description: Returns an object of type T .
ToDoubleBiFunction<T, U>	double applyAsDouble(T <i>tVal</i> , U <i>uVal</i>) Description: Acts on <i>tVal</i> and <i>uVal</i> and returns the result as a double .
ToDoubleFunction<T>	double applyAsDouble(T <i>val</i>) Description: Acts on <i>val</i> and returns the result as a double .

Table 19-19 Functional Interfaces Defined by `java.util.function` and Their Abstract Methods (*continued*)

Interface	Abstract Method
ToIntBiFunction<T, U>	int applyAsInt(T <i>tVal</i> , U <i>uVal</i>) Description: Acts on <i>tVal</i> and <i>uVal</i> and returns the result as an int .
ToIntFunction<T>	int applyAsInt(T <i>val</i>) Description: Acts on <i>val</i> and returns the result as an int .
ToLongBiFunction<T, U>	long applyAsLong(T <i>tVal</i> , U <i>uVal</i>) Description: Acts on <i>tVal</i> and <i>uVal</i> and returns the result as a long .
ToLongFunction<T>	long applyAsLong(T <i>val</i>) Description: Acts on <i>val</i> and returns the result as a long .
UnaryOperator<T>	T apply(T <i>val</i>) Description: Acts on <i>val</i> and returns the result

Table 19-19 Functional Interfaces Defined by **java.util.function** and Their Abstract Methods (*continued*)

java.util.jar

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

java.util.logging

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

java.util.prefs

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

java.util.regex

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

java.util.spi

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

java.util.stream

The **java.util.stream** package contains Java's stream API, which was added by JDK 8. A discussion of the stream API is found in Chapter 29.

java.util.zip

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

This page has been intentionally left blank

CHAPTER

20

Input/Output: Exploring `java.io`

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

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

NOTE The stream-based I/O system packaged in `java.io` and described in this chapter has been part of Java since its original release and is widely used. However, beginning with version 1.4, a second I/O system was added to Java. It is called NIO (which was originally an acronym for New I/O). NIO is packaged in `java.nio` and its subpackages. The NIO system is described in Chapter 21.

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

The I/O Classes and Interfaces

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

<code>BufferedInputStream</code>	<code>FileWriter</code>	<code>PipedOutputStream</code>
<code>BufferedOutputStream</code>	<code>FilterInputStream</code>	<code>PipedReader</code>

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

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

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

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

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

File

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

NOTE The **Path** interface and **Files** class, which are part of the NIO system, offer a powerful alternative to **File** in many cases. See Chapter 21 for details.

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

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

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

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

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

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

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

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

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

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

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

        p("File Name: " + f1.getName());
        p("Path: " + f1.getPath());
        p("Abs Path: " + f1.getAbsoluteFilePath());
        p("Parent: " + f1.getParent());
        p(f1.exists() ? "exists" : "does not exist");
    }
}
```

```

        p(f1.canWrite() ? "is writeable" : "is not writeable");
        p(f1.canRead() ? "is readable" : "is not readable");
        p("is " + (f1.isDirectory() ? "" : "not" + " a directory"));
        p(f1.isFile() ? "is normal file" : "might be a named pipe");
        p(f1.isAbsolute() ? "is absolute" : "is not absolute");
        p("File last modified: " + f1.lastModified());
        p("File size: " + f1.length() + " Bytes");
    }
}

```

This program will produce output similar to this:

```

File Name: COPYRIGHT
Path: \java\COPYRIGHT
Abs Path: C:\java\COPYRIGHT
Parent: \java
exists
is writeable
is readable
is not a directory
is normal file
is not absolute
File last modified: 1282832030047
File size: 695 Bytes

```

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

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

```
boolean renameTo(File newName)
```

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

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

```
boolean delete()
```

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

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

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

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

JDK 7 added a method to **File** called **toPath()**, which is shown here:

Path **toPath()**

toPath() returns a **Path** object that represents the file encapsulated by the invoking **File** object. (In other words, **toPath()** converts a **File** into a **Path**.) **Path** is packaged in **java.nio.file** and is part of NIO. Thus, **toPath()** forms a bridge between the older **File** class and the newer **Path** interface. (See Chapter 21 for a discussion of **Path**.)

Directories

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

String[] **list()**

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

The program shown here illustrates how to use **list()** to examine the contents of a directory:

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

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

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

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

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

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

Using `FilenameFilter`

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

String[] list(FilenameFilter *FFObj*)

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

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

boolean accept(File directory, String filename)

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

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

```
import java.io.*;

public class OnlyExt implements FilenameFilter {
    String ext;

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

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

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

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

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

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

The **listFiles()** Alternative

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

```
File[ ] listFiles()
File[ ] listFiles(FilenameFilter FObj)
File[ ] listFiles(FileFilter FObj)
```

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

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

```
boolean accept(File path)
```

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

Creating Directories

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

The AutoCloseable, Closeable, and Flushable Interfaces

There are three interfaces that are quite important to the stream classes. Two are `Closeable` and `Flushable`. They are defined in `java.io` and were added by JDK 5. The third, `AutoCloseable`, was added by JDK 7. It is packaged in `java.lang`.

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

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

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

The `Closeable` interface also defines the `close()` method. Objects of a class that implement `Closeable` can be closed. Beginning with JDK 7, `Closeable` extends `AutoCloseable`. Therefore, any class that implements `Closeable` also implements `AutoCloseable`.

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

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

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

I/O Exceptions

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

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

Two Ways to Close a Stream

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

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

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

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

The second approach to closing a stream is to automate the process by using the **try-with-resources** statement that was added by JDK 7 (and, of course, supported by JDK 8). The **try-with-resources** statement is an enhanced form of **try** that has the following form:

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

Here, *resource-specification* is a statement or statements that declares and initializes a resource, such as a file or other stream-related resource. It consists of a variable declaration in which the variable is initialized with a reference to the object being managed. When the

try block ends, the resource is automatically released. In the case of a file, this means that the file is automatically closed. Thus, there is no need to call **close()** explicitly.

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

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

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

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

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

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

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

The Stream Classes

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

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

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

The Byte Streams

The byte stream classes provide a rich environment for handling byte-oriented I/O. A byte stream can be used with any type of object, including binary data. This versatility makes byte streams important to many types of programs. Since the byte stream classes are topped by **InputStream** and **OutputStream**, our discussion begins with them.

InputStream

InputStream is an abstract class that defines Java's model of streaming byte input. It implements the **AutoCloseable** and **Closeable** interfaces. Most of the methods in this class will throw an **IOException** when an I/O error occurs. (The exceptions are **mark()** and **markSupported()**.) Table 20-1 shows the methods in **InputStream**.

NOTE Most of the methods described in Table 20-1 are implemented by the subclasses of **InputStream**.

The **mark()** and **reset()** methods are exceptions; notice their use, or lack thereof, by each subclass in the discussions that follow.

OutputStream

OutputStream is an abstract class that defines streaming byte output. It implements the **AutoCloseable**, **Closeable**, and **Flushable** interfaces. Most of the methods defined by this class return **void** and throw an **IOException** in the case of I/O errors. Table 20-2 shows the methods in **OutputStream**.

Method	Description
<code>int available()</code>	Returns the number of bytes of input currently available for reading.
<code>void close()</code>	Closes the input source. Further read attempts will generate an IOException .
<code>void mark(int numBytes)</code>	Places a mark at the current point in the input stream that will remain valid until <i>numBytes</i> bytes are read.
<code>boolean markSupported()</code>	Returns true if mark() / reset() are supported by the invoking stream.
<code>int read()</code>	Returns an integer representation of the next available byte of input. -1 is returned when the end of the file is encountered.
<code>int read(byte buffer[])</code>	Attempts to read up to <i>buffer.length</i> bytes into <i>buffer</i> and returns the actual number of bytes that were successfully read. -1 is returned when the end of the file is encountered.
<code>int read(byte buffer[], int offset, int numBytes)</code>	Attempts to read up to <i>numBytes</i> bytes into <i>buffer</i> starting at <i>buffer[offset]</i> , returning the number of bytes successfully read. -1 is returned when the end of the file is encountered.
<code>void reset()</code>	Resets the input pointer to the previously set mark.
<code>long skip(long numBytes)</code>	Ignores (that is, skips) <i>numBytes</i> bytes of input, returning the number of bytes actually ignored.

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

Method	Description
void close()	Closes the output stream. Further write attempts will generate an IOException .
void flush()	Finalizes the output state so that any buffers are cleared. That is, it flushes the output buffers.
void write(int <i>b</i>)	Writes a single byte to an output stream. Note that the parameter is an int , which allows you to call write() with an expression without having to cast it back to byte .
void write(byte <i>buffer</i> [])	Writes a complete array of bytes to an output stream.
void write(byte <i>buffer</i> [], int <i>offset</i> , int <i>numBytes</i>)	Writes a subrange of <i>numBytes</i> bytes from the array <i>buffer</i> , beginning at <i>buffer[offset]</i> .

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

FileInputStream

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

```
FileInputStream(String filePath)
FileInputStream(File fileObj)
```

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

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

```
FileInputStream f0 = new FileInputStream("/autoexec.bat")
File f = new File("/autoexec.bat");
FileInputStream f1 = new FileInputStream(f);
```

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

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

```
// Demonstrate FileInputStream.
// This program uses try-with-resources. It requires JDK 7 or later.

import java.io.*;
```

```

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

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

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

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

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

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

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

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

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

```

Here is the output produced by this program:

```

Total Available Bytes: 1785
First 44 bytes of the file one read() at a time
// Demonstrate FileInputStream.
// This pr
Still Available: 1741

```

```

Reading the next 44 with one read(b[])
ogram uses try-with-resources. It requires J

Still Available: 1697
Skipping half of remaining bytes with skip()
Still Available: 849
Reading 22 into the end of array
ogram uses try-with-rebyte[n];
    if (
        Still Available: 827

```

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

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

FileOutputStream

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

```

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

```

They can throw a **FileNotFoundException**. Here, *filePath* is the full path name of a file, and *fileObj* is a **File** object that describes the file. If *append* is **true**, the file is opened in append mode.

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

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

```

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

import java.io.*;

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

```

```
byte buf[] = source.getBytes();
FileOutputStream f0 = null;
FileOutputStream f1 = null;
FileOutputStream f2 = null;

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

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

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

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

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

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

Next, **file2.txt**:

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

Finally, **file3.txt**:

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

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

```
// Demonstrate FileOutputStream.
// This version uses try-with-resources. It requires JDK 7 or later.

import java.io.*;

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

        // Use try-with-resources to close the files.
        try (FileOutputStream f0 = new FileOutputStream("file1.txt");
             FileOutputStream f1 = new FileOutputStream("file2.txt");
             FileOutputStream f2 = new FileOutputStream("file3.txt") )
        {
            // write to first file
            for (int i=0; i < buf.length; i += 2) f0.write(buf[i]);

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

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

ByteArrayInputStream

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

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

Here, `array` is the input source. The second constructor creates an **InputStream** from a subset of the byte array that begins with the character at the index specified by `start` and is `numBytes` long.

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

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

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

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

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

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

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

```
import java.io.*;

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

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

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

```
abc
ABC
```

ByteArrayOutputStream

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

```
ByteArrayOutputStream()
ByteArrayOutputStream(int numBytes)
```

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

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

The following example demonstrates **ByteArrayOutputStream**:

```
// Demonstrate ByteArrayOutputStream.
// This program uses try-with-resources. It requires JDK 7 or later.

import java.io.*;

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

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

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

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

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

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

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

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

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

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

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

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

Filtered Byte Streams

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

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

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

Buffered Byte Streams

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

BufferedInputStream

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

BufferedInputStream has two constructors:

```
BufferedInputStream(InputStream inputStream)
```

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

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

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

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

```
// Use buffered input.
// This program uses try-with-resources. It requires JDK 7 or later.

import java.io.*;

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

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

        // Use try-with-resources to manage the file.
        try (BufferedInputStream f = new BufferedInputStream(in))
```

```
{  
    while ((c = f.read()) != -1) {  
        switch(c) {  
            case '&':  
                if (!marked) {  
                    f.mark(32);  
                    marked = true;  
                } else {  
                    marked = false;  
                }  
                break;  
            case ';':  
                if (marked) {  
                    marked = false;  
                    System.out.print("(c)");  
                } else  
                    System.out.print((char) c);  
                break;  
            case ' ':  
                if (marked) {  
                    marked = false;  
                    f.reset();  
                    System.out.print("&");  
                } else  
                    System.out.print((char) c);  
                break;  
            default:  
                if (!marked)  
                    System.out.print((char) c);  
                break;  
        }  
    }  
}  
} catch(IOException e) {  
    System.out.println("I/O Error: " + e);  
}  
}
```

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

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

BufferedOutputStream

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

Unlike buffered input, buffering output does not provide additional functionality. Buffers for output in Java are there to increase performance. Here are the two available constructors:

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

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

PushbackInputStream

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

PushbackInputStream has the following constructors:

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

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

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

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

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

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

```
// Demonstrate unread().
// This program uses try-with-resources. It requires JDK 7 or later.

import java.io.*;

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

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

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

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

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

SequenceInputStream

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

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

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

Here is a simple example that uses a **SequenceInputStream** to output the contents of two files. For demonstration purposes, this program uses the traditional technique used to

close a file. As an exercise, you might want to try changing it to use the **try-with-resources** statement.

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

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

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

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

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

    public FileInputStream nextElement() {
        try {
            return new FileInputStream(files.nextElement().toString());
        } catch (IOException e) {
            return null;
        }
    }
}

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

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

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

```
        } catch(IOException e) {
            System.out.println("Error Closing SequenceInputStream");
        }
    }
}
```

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

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

PrintStream

The **PrintStream** class provides all of the output capabilities we have been using from the **System** file handle, **System.out**, since the beginning of the book. This makes **PrintStream** one of Java's most often used classes. It implements the **Appendable**, **AutoCloseable**, **Closeable**, and **Flushable** interfaces.

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

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

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

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

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

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

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

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

Somewhat recently (with the release of JDK 5), the **printf()** method was added to **PrintStream**. It allows you to specify the precise format of the data to be written. The **printf()** method uses the **Formatter** class (described in Chapter 19) to format data. It then writes this data to the invoking stream. Although formatting can be done manually, by using **Formatter** directly, **printf()** streamlines the process. It also parallels the C/C++ **printf()** function, which makes it easy to convert existing C/C++ code into Java. Frankly, **printf()** was a much welcome addition to the Java API because it greatly simplified the output of formatted data to the console.

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

```
PrintStream printf(String fmtString, Object ... args)
PrintStream printf(Locale loc, String fmtString, Object ... args)
```

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

In general, **printf()** works in a manner similar to the **format()** method specified by **Formatter**. The *fmtString* consists of two types of items. The first type is composed of characters that are simply copied to the output buffer. The second type contains format specifiers that define the way the subsequent arguments, specified by *args*, are displayed. For complete information on formatting output, including a description of the format specifiers, see the **Formatter** class in Chapter 19.

Because **System.out** is a **PrintStream**, you can call **printf()** on **System.out**. Thus, **printf()** can be used in place of **println()** when writing to the console whenever formatted output is desired. For example, the following program uses **printf()** to output numeric values in various formats. Prior to JDK 5, such formatting required a bit of work. With the addition of **printf()**, this is now an easy task.

```
// Demonstrate printf().

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

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

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

```
System.out.printf("Negative floating-point default: %,f\n",
                  -1234567.123);
System.out.printf("Negative floating-point option: %,(f\n",
                  -1234567.123);

System.out.println();

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

The output is shown here:

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

```
Various integer formats: 3 (3) +3 00003
```

```
Default floating-point format: 1234567.123000
```

```
Floating-point with commas: 1,234,567.123000
```

```
Negative floating-point default: -1,234,567.123000
```

```
Negative floating-point option: (1,234,567.123000)
```

```
Line up positive and negative values:
```

```
 1,234,567.12
```

```
-1,234,567.12
```

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

```
PrintStream format(String fmtString, Object ... args)
```

```
PrintStream format(Locale loc, String fmtString, Object ... args)
```

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

DataOutputStream and DataInputStream

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

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

```
DataOutputStream(OutputStream outputStream)
```

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

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

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

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

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

```
DataInputStream(InputStream inputStream)
```

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

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

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

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

```
// Demonstrate DataInputStream and DataOutputStream.
// This program uses try-with-resources. It requires JDK 7 or later.

import java.io.*;

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

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

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

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

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

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

The output is shown here:

Here are the values: 98.6 1000 true

RandomAccessFile

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

`RandomAccessFile(File fileObj, String access)
throws FileNotFoundException`

`RandomAccessFile(String filename, String access)
throws FileNotFoundException`

In the first form, `fileObj` specifies the file to open as a `File` object. In the second form, the name of the file is passed in `filename`. In both cases, `access` determines what type of file access is permitted. If it is "r", then the file can be read, but not written. If it is "rw", then the file is opened in read-write mode. If it is "rws", the file is opened for read-write operations and

every change to the file's data or metadata will be immediately written to the physical device. If it is "rwd", the file is opened for read-write operations and every change to the file's data will be immediately written to the physical device.

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

```
void seek(long newPos) throws IOException
```

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

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

```
void setLength(long len) throws IOException
```

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

The Character Streams

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

Reader

Reader is an abstract class that defines Java's model of streaming character input. It implements the **AutoCloseable**, **Closeable**, and **Readable** interfaces. All of the methods in this class (except for **markSupported()**) will throw an **IOException** on error conditions. Table 20-3 provides a synopsis of the methods in **Reader**.

Writer

Writer is an abstract class that defines streaming character output. It implements the **AutoCloseable**, **Closeable**, **Flushable**, and **Appendable** interfaces. All of the methods in this class throw an **IOException** in the case of errors. Table 20-4 shows a synopsis of the methods in **Writer**.

Method	Description
abstract void close()	Closes the input source. Further read attempts will generate an IOException .
void mark(int numChars)	Places a mark at the current point in the input stream that will remain valid until <i>numChars</i> characters are read.
boolean markSupported()	Returns true if mark() / reset() are supported on this stream.
int read()	Returns an integer representation of the next available character from the invoking input stream. -1 is returned when the end of the file is encountered.
int read(char <i>buffer</i> [])	Attempts to read up to <i>buffer.length</i> characters into <i>buffer</i> and returns the actual number of characters that were successfully read. -1 is returned when the end of the file is encountered.
int read(CharBuffer <i>buffer</i>)	Attempts to read characters into <i>buffer</i> and returns the actual number of characters that were successfully read. -1 is returned when the end of the file is encountered.
abstract int read(char <i>buffer</i> [], int <i>offset</i> , int <i>numChars</i>)	Attempts to read up to <i>numChars</i> characters into <i>buffer</i> starting at <i>buffer[offset]</i> , returning the number of characters successfully read. -1 is returned when the end of the file is encountered.
boolean ready()	Returns true if the next input request will not wait. Otherwise, it returns false .
void reset()	Resets the input pointer to the previously set mark.
long skip(long <i>numChars</i>)	Skips over <i>numChars</i> characters of input, returning the number of characters actually skipped.

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

Method	Description
Writer append(char <i>ch</i>)	Appends <i>ch</i> to the end of the invoking output stream. Returns a reference to the invoking stream.
Writer append(CharSequence <i>chars</i>)	Appends <i>chars</i> to the end of the invoking output stream. Returns a reference to the invoking stream.
Writer append(CharSequence <i>chars</i> , int <i>begin</i> , int <i>end</i>)	Appends the subrange of <i>chars</i> specified by <i>begin</i> and <i>end</i> -1 to the end of the invoking output stream. Returns a reference to the invoking stream.
abstract void close()	Closes the output stream. Further write attempts will generate an IOException .
abstract void flush()	Finalizes the output state so that any buffers are cleared. That is, it flushes the output buffers.

Table 20-4 The Methods Defined by **Writer**

Method	Description
void write(int <i>ch</i>)	Writes a single character to the invoking output stream. Note that the parameter is an int , which allows you to call write with an expression without having to cast it back to char . However, only the low-order 16 bits are written.
void write(char <i>buffer</i> [])	Writes a complete array of characters to the invoking output stream.
abstract void write(char <i>buffer</i> [], int <i>offset</i> , int <i>numChars</i>)	Writes a subrange of <i>numChars</i> characters from the array <i>buffer</i> , beginning at <i>buffer[offset]</i> to the invoking output stream.
void write(String <i>str</i>)	Writes <i>str</i> to the invoking output stream.
void write(String <i>str</i> , int <i>offset</i> , int <i>numChars</i>)	Writes a subrange of <i>numChars</i> characters from the string <i>str</i> , beginning at the specified <i>offset</i> .

Table 20-4 The Methods Defined by **Writer** (*continued*)

FileReader

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

```
FileReader(String filePath)
FileReader(File fileObj)
```

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

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

```
// Demonstrate FileReader.
// This program uses try-with-resources. It requires JDK 7 or later.

import java.io.*;

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

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

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

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

FileWriter

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

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

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

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

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

```
// Demonstrate FileWriter.
// This program uses try-with-resources. It requires JDK 7 or later.

import java.io.*;

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

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

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

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

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

CharArrayReader

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

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

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

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

The following example uses a pair of **CharArrayReaders**:

```
// Demonstrate CharArrayReader.  
// This program uses try-with-resources. It requires JDK 7 or later.  
  
import java.io.*;  
  
public class CharArrayReaderDemo {  
    public static void main(String args[]) {  
        String tmp = "abcdefghijklmnopqrstuvwxyz";  
        int length = tmp.length();  
        char c[] = new char[length];  
  
        tmp.getChars(0, length, c, 0);  
        int i;  
  
        try (CharArrayReader input1 = new CharArrayReader(c) )  
        {  
            System.out.println("input1 is:");  
            while((i = input1.read()) != -1) {  
                System.out.print((char)i);  
            }  
            System.out.println();  
        } catch(IOException e) {  
            System.out.println("I/O Error: " + e);  
        }  
  
        try ( CharArrayReader input2 = new CharArrayReader(c, 0, 5) )  
        {  
            System.out.println("input2 is:");  
            while((i = input2.read()) != -1) {  
                System.out.print((char)i);  
            }  
        }  
    }  
}
```

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

The **input1** object is constructed using the entire lowercase alphabet, whereas **input2** contains only the first five letters. Here is the output:

```
input1 is:  
abcdefghijklmnopqrstuvwxyz  
input2 is:  
abcde
```

CharArrayWriter

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

CharArrayWriter()

CharArrayWriter(int numChars)

In the first form, a buffer with a default size is created. In the second, a buffer is created with a size equal to that specified by *numChars*. The buffer is held in the **buf** field of **CharArrayWriter**. The buffer size will be increased automatically, if needed. The number of characters held by the buffer is contained in the **count** field of **CharArrayWriter**. Both **buf** and **count** are protected fields.

The `close()` method has no effect on a `CharArrayWriter`.

The following example demonstrates **CharArrayWriter** by reworking the sample program shown earlier for **ByteArrayOutputStream**. It produces the same output as the previous version.

```
// Demonstrate CharArrayWriter.  
// This program uses try-with-resources. It requires JDK 7 or later.  
  
import java.io.*;  
  
class CharArrayWriterDemo {  
    public static void main(String args[]) throws IOException {  
        CharArrayWriter f = new CharArrayWriter();  
        String s = "This should end up in the array";  
        char buf[] = new char[s.length()];  
  
        s.getChars(0, s.length(), buf, 0);  
  
        try {  
            f.write(buf);  
        } catch(IOException e) {  
            System.out.println("Error Writing to Buffer");  
            return;  
        }  
    }  
}
```

```

System.out.println("Buffer as a string");
System.out.println(f.toString());
System.out.println("Into array");

char c[] = f.toCharArray();
for (int i=0; i<c.length; i++) {
    System.out.print(c[i]);
}

System.out.println("\nTo a FileWriter()");

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

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

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

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

```

BufferedReader

BufferedReader improves performance by buffering input. It has two constructors:

```

BufferedReader(Reader inputStream)
BufferedReader(Reader inputStream, int bufSize)

```

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

Closing a **BufferedReader** also causes the underlying stream specified by *inputStream* to be closed.

As is the case with the byte-oriented stream, buffering an input character stream also provides the foundation required to support moving backward in the stream within the available buffer. To support this, **BufferedReader** implements the **mark()** and **reset()** methods, and **BufferedReader.markSupported()** returns **true**. JDK 8 adds a new method to **BufferedReader** called **lines()**. It returns a **Stream** reference to the sequence of lines read by the reader. (**Stream** is part of the new stream API discussed in Chapter 29.)

The following example reworks the **BufferedInputStream** example, shown earlier, so that it uses a **BufferedReader** character stream rather than a buffered byte stream. As before, it uses the **mark()** and **reset()** methods to parse a stream for the HTML entity reference for the copyright symbol. Such a reference begins with an ampersand (&) and ends with a semicolon (;) without any intervening whitespace. The sample input has two ampersands to show the case where the **reset()** happens and where it does not. Output is the same as that shown earlier.

```
// Use buffered input.  
// This program uses try-with-resources. It requires JDK 7 or later.  
  
import java.io.*;  
  
class BufferedReaderDemo {  
    public static void main(String args[]) throws IOException {  
        String s = "This is a &copy; copyright symbol " +  
            "but this is &copy not.\n";  
        char buf[] = new char[s.length()];  
        s.getChars(0, s.length(), buf, 0);  
  
        CharArrayReader in = new CharArrayReader(buf);  
        int c;  
        boolean marked = false;  
  
        try ( BufferedReader f = new BufferedReader(in) )  
        {  
  
            while ((c = f.read()) != -1) {  
                switch(c) {  
                    case '&':  
                        if (!marked) {  
                            f.mark(32);  
                            marked = true;  
                        } else {  
                            marked = false;  
                        }  
                        break;  
                    case ';':  
                        if (marked) {  
                            marked = false;  
                            System.out.print("(c) ");  
                        } else  
                            System.out.print((char) c);  
                        break;  
                    case ' ':  
                        if (marked) {  
                            marked = false;  
                            f.reset();  
                            System.out.print("&");  
                        } else  
                            System.out.print((char) c);  
                        break;  
                    default:  
                        if (!marked)  
                            System.out.print((char) c);  
                        break;  
                }  
            }  
        } catch(IOException e) {  
            System.out.println("I/O Error: " + e);  
        }  
    }  
}
```

BufferedWriter

A **BufferedWriter** is a **Writer** that buffers output. Using a **BufferedWriter** can improve performance by reducing the number of times data is actually physically written to the output device.

A **BufferedWriter** has these two constructors:

```
BufferedWriter(Writer outputStream)
BufferedWriter(Writer outputStream, int bufSize)
```

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

PushbackReader

The **PushbackReader** class allows one or more characters to be returned to the input stream. This allows you to look ahead in the input stream. Here are its two constructors:

```
PushbackReader(Reader inputStream)
PushbackReader(Reader inputStream, int bufSize)
```

The first form creates a buffered stream that allows one character to be pushed back. In the second, the size of the pushback buffer is passed in *bufSize*.

Closing a **PushbackReader** also closes the underlying stream specified by *inputStream*.

PushbackReader provides **unread()**, which returns one or more characters to the invoking input stream. It has the three forms shown here:

```
void unread(int ch) throws IOException
void unread(char buffer[]) throws IOException
void unread(char buffer[], int offset, int numChars) throws IOException
```

The first form pushes back the character passed in *ch*. This will be the next character returned by a subsequent call to **read()**. The second form returns the characters in *buffer*. The third form pushes back *numChars* characters beginning at *offset* from *buffer*. An **IOException** will be thrown if there is an attempt to return a character when the pushback buffer is full.

The following program reworks the earlier **PushbackInputStream** example by replacing **PushbackInputStream** with **PushbackReader**. As before, it shows how a programming language parser can use a pushback stream to deal with the difference between the == operator for comparison and the = operator for assignment.

```
// Demonstrate unread().
// This program uses try-with-resources. It requires JDK 7 or later.

import java.io.*;

class PushbackReaderDemo {
    public static void main(String args[]) {
        String s = "if (a == 4) a = 0;\n";
        char buf[] = new char[s.length()];
        s.getChars(0, s.length(), buf, 0);
        CharArrayReader in = new CharArrayReader(buf);

        int c;
```

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

PrintWriter

PrintWriter is essentially a character-oriented version of **PrintStream**. It implements the **Appendable**, **AutoCloseable**, **Closeable**, and **Flushable** interfaces. **PrintWriter** has several constructors. The following have been supplied by **PrintWriter** from the start:

```
PrintWriter(OutputStream outputStream)
PrintWriter(OutputStream outputStream, boolean autoFlushingOn)
PrintWriter(Writer outputStream)
PrintWriter(Writer outputStream, boolean autoFlushingOn)
```

Here, `outputStream` specifies an open **OutputStream** that will receive output. The `autoFlushingOn` parameter controls whether the output buffer is automatically flushed every time `println()`, `printf()`, or `format()` is called. If `autoFlushingOn` is `true`, flushing automatically takes place. If `false`, flushing is not automatic. Constructors that do not specify the `autoFlushingOn` parameter do not automatically flush.

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

```
PrintWriter(File outputFile) throws FileNotFoundException  
PrintWriter(File outputFile, String charSet)  
    throws FileNotFoundException, UnsupportedEncodingException  
PrintWriter(String outputFileName) throws FileNotFoundException  
PrintWriter(String outputFileName, String charSet)  
    throws FileNotFoundException, UnsupportedEncodingException
```

These allow a **PrintWriter** to be created from a **File** object or by specifying the name of a file. In either case, the file is automatically created. Any preexisting file by the same name is destroyed. Once created, the **PrintWriter** object directs all output to the specified file. You can specify a character encoding by passing its name in *charSet*.

PrintWriter supports the **print()** and **println()** methods for all types, including **Object**. If an argument is not a primitive type, the **PrintWriter** methods will call the object's **toString()** method and then output the result.

PrintWriter also supports the **printf()** method. It works the same way it does in the **PrintStream** class described earlier: It allows you to specify the precise format of the data. Here is how **printf()** is declared in **PrintWriter**:

```
PrintWriter printf(String fmtString, Object ... args)
PrintWriter printf(Locale loc, String fmtString, Object ...args)
```

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

The **format()** method is also supported. It has these general forms:

```
PrintWriter format(String fmtString, Object ... args)
PrintWriter format(Locale loc, String fmtString, Object ... args)
```

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

The Console Class

The **Console** class was added to **java.io** by JDK 6. It is used to read from and write to the console, if one exists. It implements the **Flushable** interface. **Console** is primarily a convenience class because most of its functionality is available through **System.in** and **System.out**. However, its use can simplify some types of console interactions, especially when reading strings from the console.

Console supplies no constructors. Instead, a **Console** object is obtained by calling **System.console()**, which is shown here:

```
static Console console()
```

If a console is available, then a reference to it is returned. Otherwise, **null** is returned. A console will not be available in all cases. Thus, if **null** is returned, no console I/O is possible.

Console defines the methods shown in Table 20-5. Notice that the input methods, such as **readLine()**, throw **IOException** if an input error occurs. **IOException** is a subclass of **Error**. It indicates an I/O failure that is beyond the control of your program. Thus, you will not normally catch an **IOException**. Frankly, if an **IOException** is thrown while accessing the console, it usually means there has been a catastrophic system failure.

Also notice the **readPassword()** methods. These methods let your application read a password without echoing what is typed. When reading passwords, you should "zero-out" both the array that holds the string entered by the user and the array that holds the password that the string is tested against. This reduces the chance that a malicious program will be able to obtain a password by scanning memory.

Method	Description
void flush()	Causes buffered output to be written physically to the console.
Console format(String <i>fmtString</i> , Object... <i>args</i>)	Writes <i>args</i> to the console using the format specified by <i>fmtString</i> .
Console printf(String <i>fmtString</i> , Object... <i>args</i>)	Writes <i>args</i> to the console using the format specified by <i>fmtString</i> .
Reader reader()	Returns a reference to a Reader connected to the console.
String readLine()	Reads and returns a string entered at the keyboard. Input stops when the user presses ENTER. If the end of the console input stream has been reached, null is returned. An IOError is thrown on failure.
String readLine(String <i>fmtString</i> , Object... <i>args</i>)	Displays a prompting string formatted as specified by <i>fmtString</i> and <i>args</i> , and then reads and returns a string entered at the keyboard. Input stops when the user presses ENTER. If the end of the console input stream has been reached, null is returned. An IOError is thrown on failure.
char[] readPassword()	Reads a string entered at the keyboard. Input stops when the user presses ENTER. The string is not displayed. If the end of the console input stream has been reached, null is returned. An IOError is thrown on failure.
char[] readPassword(String <i>fmtString</i> , Object... <i>args</i>)	Displays a prompting string formatted as specified by <i>fmtString</i> and <i>args</i> , and then reads a string entered at the keyboard. Input stops when the user presses ENTER. The string is not displayed. If the end of the console input stream has been reached, null is returned. An IOError is thrown on failure.
PrintWriter writer()	Returns a reference to a Writer connected to the console.

Table 20-5 The Methods Defined by **Console**

Here is an example that demonstrates the **Console** class:

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

class ConsoleDemo {
    public static void main(String args[]) {
        String str;
        Console con;
```

```
// Obtain a reference to the console.  
con = System.console();  
// If no console available, exit.  
if(con == null) return;  
  
// Read a string and then display it.  
str = con.readLine("Enter a string: ");  
con.printf("Here is your string: %s\n", str);  
}  
}
```

Here is sample output:

```
Enter a string: This is a test.  
Here is your string: This is a test.
```

Serialization

Serialization is the process of writing the state of an object to a byte stream. This is useful when you want to save the state of your program to a persistent storage area, such as a file. At a later time, you may restore these objects by using the process of *deserialization*.

Serialization is also needed to implement *Remote Method Invocation (RMI)*. RMI allows a Java object on one machine to invoke a method of a Java object on a different machine. An object may be supplied as an argument to that remote method. The sending machine serializes the object and transmits it. The receiving machine deserializes it. (More information about RMI appears in Chapter 30.)

Assume that an object to be serialized has references to other objects, which, in turn, have references to still more objects. This set of objects and the relationships among them form a directed graph. There may also be circular references within this object graph. That is, object X may contain a reference to object Y, and object Y may contain a reference back to object X. Objects may also contain references to themselves. The object serialization and deserialization facilities have been designed to work correctly in these scenarios. If you attempt to serialize an object at the top of an object graph, all of the other referenced objects are recursively located and serialized. Similarly, during the process of deserialization, all of these objects and their references are correctly restored.

An overview of the interfaces and classes that support serialization follows.

Serializable

Only an object that implements the **Serializable** interface can be saved and restored by the serialization facilities. The **Serializable** interface defines no members. It is simply used to indicate that a class may be serialized. If a class is serializable, all of its subclasses are also serializable.

Variables that are declared as **transient** are not saved by the serialization facilities. Also, **static** variables are not saved.

Externalizable

The Java facilities for serialization and deserialization have been designed so that much of the work to save and restore the state of an object occurs automatically. However, there are cases in which the programmer may need to have control over these processes. For example, it may be desirable to use compression or encryption techniques. The **Externalizable** interface is designed for these situations.

The **Externalizable** interface defines these two methods:

```
void readExternal(ObjectInput inStream)
throws IOException, ClassNotFoundException
void writeExternal(ObjectOutput outStream)
throws IOException
```

In these methods, *inStream* is the byte stream from which the object is to be read, and *outStream* is the byte stream to which the object is to be written.

ObjectOutput

The **ObjectOutput** interface extends the **DataOutput** and **AutoCloseable** interfaces and supports object serialization. It defines the methods shown in Table 20-6. Note especially the **writeObject()** method. This is called to serialize an object. All of these methods will throw an **IOException** on error conditions.

Method	Description
void close()	Closes the invoking stream. Further write attempts will generate an IOException .
void flush()	Finalizes the output state so any buffers are cleared. That is, it flushes the output buffers.
void write(byte <i>buffer</i> [])	Writes an array of bytes to the invoking stream.
void write(byte <i>buffer</i> [], int <i>offset</i> , int <i>numBytes</i>)	Writes a subrange of <i>numBytes</i> bytes from the array <i>buffer</i> , beginning at <i>buffer</i> [<i>offset</i>].
void write(int <i>b</i>)	Writes a single byte to the invoking stream. The byte written is the low-order byte of <i>b</i> .
void writeObject(Object <i>obj</i>)	Writes object <i>obj</i> to the invoking stream.

Table 20-6 The Methods Defined by **ObjectOutput**

ObjectOutputStream

The **ObjectOutputStream** class extends the **OutputStream** class and implements the **ObjectOutput** interface. It is responsible for writing objects to a stream. One constructor of this class is shown here:

```
ObjectOutputStream(OutputStream outStream) throws IOException
```

The argument *outStream* is the output stream to which serialized objects will be written. Closing an **ObjectOutputStream** automatically closes the underlying stream specified by *outStream*.

Several commonly used methods in this class are shown in Table 20-7. They will throw an **IOException** on error conditions. There is also an inner class to **ObjectOutput** called **PutField**. It facilitates the writing of persistent fields, and its use is beyond the scope of this book.

Method	Description
void close()	Closes the invoking stream. Further write attempts will generate an IOException . The underlying stream is also closed.
void flush()	Finalizes the output state so any buffers are cleared. That is, it flushes the output buffers.
void write(byte <i>buffer</i> [])	Writes an array of bytes to the invoking stream.
void write(byte <i>buffer</i> [], int <i>offset</i> , int <i>numBytes</i>)	Writes a subrange of <i>numBytes</i> bytes from the array <i>buffer</i> , beginning at <i>buffer</i> [<i>offset</i>].
void write(int <i>b</i>)	Writes a single byte to the invoking stream. The byte written is the low-order byte of <i>b</i> .
void writeBoolean(boolean <i>b</i>)	Writes a boolean to the invoking stream.
void writeByte(int <i>b</i>)	Writes a byte to the invoking stream. The byte written is the low-order byte of <i>b</i> .
void writeBytes(String <i>str</i>)	Writes the bytes representing <i>str</i> to the invoking stream.
void writeChar(int <i>c</i>)	Writes a char to the invoking stream.
void writeChars(String <i>str</i>)	Writes the characters in <i>str</i> to the invoking stream.
void writeDouble(double <i>d</i>)	Writes a double to the invoking stream.
void writeFloat(float <i>f</i>)	Writes a float to the invoking stream.
void writeInt(int <i>i</i>)	Writes an int to the invoking stream.
void writeLong(long <i>l</i>)	Writes a long to the invoking stream.
final void writeObject(Object <i>obj</i>)	Writes <i>obj</i> to the invoking stream.
void writeShort(int <i>i</i>)	Writes a short to the invoking stream.

Table 20-7 A Sampling of Commonly Used Methods Defined by **ObjectOutputStream**

ObjectInput

The **ObjectInput** interface extends the **DataInput** and **AutoCloseable** interfaces and defines the methods shown in Table 20-8. It supports object serialization. Note especially the **readObject()** method. This is called to deserialize an object. All of these methods will throw an **IOException** on error conditions. The **readObject()** method can also throw **ClassNotFoundException**.

ObjectInputStream

The **ObjectInputStream** class extends the **InputStream** class and implements the **ObjectInput** interface. **ObjectInputStream** is responsible for reading objects from a stream. One constructor of this class is shown here:

```
ObjectInputStream(InputStream inStream) throws IOException
```

The argument *inStream* is the input stream from which serialized objects should be read. Closing an **ObjectInputStream** automatically closes the underlying stream specified by *inStream*.

Several commonly used methods in this class are shown in Table 20-9. They will throw an **IOException** on error conditions. The **readObject()** method can also throw **ClassNotFoundException**. There is also an inner class to **ObjectInputStream** called **GetField**. It facilitates the reading of persistent fields, and its use is beyond the scope of this book.

Method	Description
<code>int available()</code>	Returns the number of bytes that are now available in the input buffer.
<code>void close()</code>	Closes the invoking stream. Further read attempts will generate an IOException .
<code>int read()</code>	Returns an integer representation of the next available byte of input. <code>-1</code> is returned when the end of the file is encountered.
<code>int read(byte <i>buffer</i>[])</code>	Attempts to read up to <code>buffer.length</code> bytes into <i>buffer</i> , returning the number of bytes that were successfully read. <code>-1</code> is returned when the end of the file is encountered.
<code>int read(byte <i>buffer</i>[], int <i>offset</i>, int <i>numBytes</i>)</code>	Attempts to read up to <code>numBytes</code> bytes into <i>buffer</i> starting at <code>buffer[offset]</code> , returning the number of bytes that were successfully read. <code>-1</code> is returned when the end of the file is encountered.
<code>Object readObject()</code>	Reads an object from the invoking stream.
<code>long skip(long <i>numBytes</i>)</code>	Ignores (that is, skips) <code>numBytes</code> bytes in the invoking stream, returning the number of bytes actually ignored.

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

Method	Description
int available()	Returns the number of bytes that are now available in the input buffer.
void close()	Closes the invoking stream. Further read attempts will generate an IOException . The underlying stream is also closed.
int read()	Returns an integer representation of the next available byte of input. -1 is returned when the end of the file is encountered.
int read(byte <i>buffer</i> [], int <i>offset</i> , int <i>numBytes</i>)	Attempts to read up to <i>numBytes</i> bytes into <i>buffer</i> starting at <i>buffer[offset]</i> , returning the number of bytes successfully read. -1 is returned when the end of the file is encountered.
Boolean readBoolean()	Reads and returns a boolean from the invoking stream.
byte readByte()	Reads and returns a byte from the invoking stream.
char readChar()	Reads and returns a char from the invoking stream.
double readDouble()	Reads and returns a double from the invoking stream.
float readFloat()	Reads and returns a float from the invoking stream.
void readFully(byte <i>buffer</i> [])	Reads <i>buffer.length</i> bytes into <i>buffer</i> . Returns only when all bytes have been read.
void readFully(byte <i>buffer</i> [], int <i>offset</i> , int <i>numBytes</i>)	Reads <i>numBytes</i> bytes into <i>buffer</i> starting at <i>buffer[offset]</i> . Returns only when <i>numBytes</i> have been read.
int readInt()	Reads and returns an int from the invoking stream.
long readLong()	Reads and returns a long from the invoking stream.
final Object readObject()	Reads and returns an object from the invoking stream.
short readShort()	Reads and returns a short from the invoking stream.
int readUnsignedByte()	Reads and returns an unsigned byte from the invoking stream.
int readUnsignedShort()	Reads and returns an unsigned short from the invoking stream.

Table 20-9 Commonly Used Methods Defined by **ObjectInputStream**

A Serialization Example

The following program illustrates how to use object serialization and deserialization. It begins by instantiating an object of class **MyClass**. This object has three instance variables that are of types **String**, **int**, and **double**. This is the information we want to save and restore.

A **FileOutputStream** is created that refers to a file named "serial", and an **ObjectOutputStream** is created for that file stream. The **writeObject()** method of **ObjectOutputStream** is then used to serialize our object. The object output stream is flushed and closed.

A **FileInputStream** is then created that refers to the file named "serial", and an **ObjectInputStream** is created for that file stream. The **readObject()** method of **ObjectInputStream** is then used to deserialize our object. The object input stream is then closed.

Note that **MyClass** is defined to implement the **Serializable** interface. If this is not done, a **NotSerializableException** is thrown. Try experimenting with this program by declaring some of the **MyClass** instance variables to be **transient**. That data is then not saved during serialization.

```
// A serialization demo.  
// This program uses try-with-resources. It requires JDK 7 or later.  
  
import java.io.*;  
  
public class SerializationDemo {  
    public static void main(String args[]) {  
  
        // Object serialization  
  
        try ( ObjectOutputStream objOStrm =  
              new ObjectOutputStream(new FileOutputStream("serial")) )  
        {  
            MyClass object1 = new MyClass("Hello", -7, 2.7e10);  
            System.out.println("object1: " + object1);  
  
            objOStrm.writeObject(object1);  
        }  
        catch(IOException e) {  
            System.out.println("Exception during serialization: " + e);  
        }  
  
        // Object deserialization  
  
        try ( ObjectInputStream objIStrm =  
              new ObjectInputStream(new FileInputStream("serial")) )  
        {  
            MyClass object2 = (MyClass)objIStrm.readObject();  
            System.out.println("object2: " + object2);  
        }  
        catch(Exception e) {  
            System.out.println("Exception during deserialization: " + e);  
        }  
    }  
  
    class MyClass implements Serializable {  
        String s;  
        int i;  
        double d;  
  
        public MyClass(String s, int i, double d) {  
            this.s = s;  
            this.i = i;  
            this.d = d;  
        }  
    }  
}
```

```
public String toString() {
    return "s=" + s + "; i=" + i + "; d=" + d;
}
}
```

This program demonstrates that the instance variables of **object1** and **object2** are identical. The output is shown here:

```
object1: s=Hello; i=-7; d=2.7E10
object2: s=Hello; i=-7; d=2.7E10
```

Stream Benefits

The streaming interface to I/O in Java provides a clean abstraction for a complex and often cumbersome task. The composition of the filtered stream classes allows you to dynamically build the custom streaming interface to suit your data transfer requirements. Java programs written to adhere to the abstract, high-level **InputStream**, **OutputStream**, **Reader**, and **Writer** classes will function properly in the future even when new and improved concrete stream classes are invented. As you will see in Chapter 22, this model works very well when we switch from a file system-based set of streams to the network and socket streams. Finally, serialization of objects plays an important role in many types of Java programs. Java's serialization I/O classes provide a portable solution to this sometimes tricky task.

CHAPTER

21

Exploring NIO

Beginning with version 1.4, Java has provided a second I/O system called NIO (which is short for *New I/O*). It supports a buffer-oriented, channel-based approach to I/O operations. With the release of JDK 7, the NIO system was greatly expanded, providing enhanced support for file-handling and file system features. In fact, so significant were the changes that the term *NIO.2* is often used. Because of the capabilities supported by the NIO file classes, NIO is expected to become an increasingly important approach to file handling. This chapter explores several of the key features of the NIO system.

The NIO Classes

The NIO classes are contained in the packages shown here:

Package	Purpose
java.nio	Top-level package for the NIO system. Encapsulates various types of buffers that contain data operated upon by the NIO system.
java.nio.channels	Supports channels, which are essentially open I/O connections.
java.nio.channels.spi	Supports service providers for channels.
java.nio.charset	Encapsulates character sets. Also supports encoders and decoders that convert characters to bytes and bytes to characters, respectively.
java.nio.charset.spi	Supports service providers for character sets.
java.nio.file	Provides support for files.
java.nio.file.attribute	Provides support for file attributes.
java.nio.file.spi	Supports service providers for file systems.

Before we begin, it is important to emphasize that the NIO subsystem does not replace the stream-based I/O classes found in **java.io**, which are discussed in Chapter 20, and good working knowledge of the stream-based I/O in **java.io** is helpful to understanding NIO.

NOTE This chapter assumes that you have read the overview of I/O given in Chapter 13 and the discussion of stream-based I/O supplied in Chapter 20.

NIO Fundamentals

The NIO system is built on two foundational items: buffers and channels. A *buffer* holds data. A *channel* represents an open connection to an I/O device, such as a file or a socket. In general, to use the NIO system, you obtain a channel to an I/O device and a buffer to hold data. You then operate on the buffer, inputting or outputting data as needed. The following sections examine buffers and channels in more detail.

Buffers

Buffers are defined in the **java.nio** package. All buffers are subclasses of the **Buffer** class, which defines the core functionality common to all buffers: current position, limit, and capacity. The *current position* is the index within the buffer at which the next read or write operation will take place. The current position is advanced by most read or write operations. The *limit* is the index value one past the last valid location in the buffer. The *capacity* is the number of elements that the buffer can hold. Often the limit equals the capacity of the buffer. **Buffer** also supports mark and reset. **Buffer** defines several methods, which are shown in Table 21-1.

Method	Description
abstract Object array()	If the invoking buffer is backed by an array, returns a reference to the array. Otherwise, an UnsupportedOperationException is thrown. If the array is read-only, a ReadOnlyBufferException is thrown.
abstract int arrayOffset()	If the invoking buffer is backed by an array, returns the index of the first element. Otherwise, an UnsupportedOperationException is thrown. If the array is read-only, a ReadOnlyBufferException is thrown.
final int capacity()	Returns the number of elements that the invoking buffer is capable of holding.
final Buffer clear()	Clears the invoking buffer and returns a reference to the buffer.
final Buffer flip()	Sets the invoking buffer's limit to the current position and resets the current position to 0. Returns a reference to the buffer.
abstract boolean hasArray()	Returns true if the invoking buffer is backed by a read/write array and false otherwise.
final boolean hasRemaining()	Returns true if there are elements remaining in the invoking buffer. Returns false otherwise.

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

Method	Description
abstract boolean isDirect()	Returns true if the invoking buffer is direct, which means I/O operations act directly upon it. Returns false otherwise.
abstract boolean isReadOnly()	Returns true if the invoking buffer is read-only. Returns false otherwise.
final int limit()	Returns the invoking buffer's limit.
final Buffer limit(int <i>n</i>)	Sets the invoking buffer's limit to <i>n</i> . Returns a reference to the buffer.
final Buffer mark()	Sets the mark and returns a reference to the invoking buffer.
final int position()	Returns the current position.
final Buffer position(int <i>n</i>)	Sets the invoking buffer's current position to <i>n</i> . Returns a reference to the buffer.
int remaining()	Returns the number of elements available before the limit is reached. In other words, it returns the limit minus the current position.
final Buffer reset()	Resets the current position of the invoking buffer to the previously set mark. Returns a reference to the buffer.
final Buffer rewind()	Sets the position of the invoking buffer to 0. Returns a reference to the buffer.

Table 21-1 The Methods Defined by **Buffer** (*continued*)

From **Buffer**, the following specific buffer classes are derived, which hold the type of data that their names imply:

ByteBuffer	CharBuffer	DoubleBuffer	FloatBuffer
IntBuffer	LongBuffer	MappedByteBuffer	ShortBuffer

MappedByteBuffer is a subclass of **ByteBuffer** and is used to map a file to a buffer.

All of the aforementioned buffers provide various **get()** and **put()** methods, which allow you to get data from a buffer or put data into a buffer. (Of course, if a buffer is read-only, then **put()** operations are not available.) Table 21-2 shows the **get()** and **put()** methods defined by **ByteBuffer**. The other buffer classes have similar methods. All buffer classes also support methods that perform various buffer operations. For example, you can allocate a buffer manually using **allocate()**. You can wrap an array inside a buffer using **wrap()**. You can create a subsequence of a buffer using **slice()**.

Channels

Channels are defined in **java.nio.channels**. A channel represents an open connection to an I/O source or destination. Channels implement the **Channel** interface. It extends **Closeable**, and it extends **AutoCloseable**. By implementing **AutoCloseable**, channels can be managed

Method	Description
abstract byte get()	Returns the byte at the current position.
ByteBuffer get(byte <i>vals</i> [])	Copies the invoking buffer into the array referred to by <i>vals</i> . Returns a reference to the buffer. If there are not <i>vals.length</i> elements remaining in the buffer, a BufferUnderflowException is thrown.
ByteBuffer get(byte <i>vals</i> [], int <i>start</i> , int <i>num</i>)	Copies <i>num</i> elements from the invoking buffer into the array referred to by <i>vals</i> , beginning at the index specified by <i>start</i> . Returns a reference to the buffer. If there are not <i>num</i> elements remaining in the buffer, a BufferUnderflowException is thrown.
abstract byte get(int <i>idx</i>)	Returns the byte at the index specified by <i>idx</i> within the invoking buffer.
abstract ByteBuffer put(byte <i>b</i>)	Copies <i>b</i> into the invoking buffer at the current position. Returns a reference to the buffer. If the buffer is full, a BufferOverflowException is thrown.
final ByteBuffer put(byte <i>vals</i> [])	Copies all elements of <i>vals</i> into the invoking buffer, beginning at the current position. Returns a reference to the buffer. If the buffer cannot hold all of the elements, a BufferOverflowException is thrown.
ByteBuffer put(byte <i>vals</i> [], int <i>start</i> , int <i>num</i>)	Copies <i>num</i> elements from <i>vals</i> , beginning at <i>start</i> , into the invoking buffer. Returns a reference to the buffer. If the buffer cannot hold all of the elements, a BufferOverflowException is thrown.
ByteBuffer put(ByteBuffer <i>bb</i>)	Copies the elements in <i>bb</i> to the invoking buffer, beginning at the current position. If the buffer cannot hold all of the elements, a BufferOverflowException is thrown. Returns a reference to the buffer.
abstract ByteBuffer put(int <i>idx</i> , byte <i>b</i>)	Copies <i>b</i> into the invoking buffer at the location specified by <i>idx</i> . Returns a reference to the buffer.

Table 21-2 The **get()** and **put()** Methods Defined for **ByteBuffer**

with a **try-with-resources** statement. When used in a **try-with-resources** block, a channel is closed automatically when it is no longer needed. (See Chapter 13 for a discussion of **try-with-resources**.)

One way to obtain a channel is by calling **getChannel()** on an object that supports channels. For example, **getChannel()** is supported by the following I/O classes:

DatagramSocket	FileInputStream	FileOutputStream
RandomAccessFile	ServerSocket	Socket

The specific type of channel returned depends upon the type of object **getChannel()** is called on. For example, when called on a **FileInputStream**, **FileOutputStream**, or **RandomAccessFile**, **getChannel()** returns a channel of type **FileChannel**. When called on a **Socket**, **getChannel()** returns a **SocketChannel**.

Another way to obtain a channel is to use one of the **static** methods defined by the **Files** class. For example, using **Files**, you can obtain a byte channel by calling **newByteChannel()**. It returns a **SeekableByteChannel**, which is an interface implemented by **FileChannel**. (The **Files** class is examined in detail later in this chapter.)

Channels such as **FileChannel** and **SocketChannel** support various **read()** and **write()** methods that enable you to perform I/O operations through the channel. For example, here are a few of the **read()** and **write()** methods defined for **FileChannel**.

Method	Description
abstract int read(ByteBuffer <i>bb</i>) throws IOException	Reads bytes from the invoking channel into <i>bb</i> until the buffer is full or there is no more input. Returns the number of bytes actually read. Returns -1 at end-of-stream.
abstract int read(ByteBuffer <i>bb</i> , long <i>start</i>) throws IOException	Beginning at the file location specified by <i>start</i> , reads bytes from the invoking channel into <i>bb</i> until the buffer is full or there is no more input. The current position is unchanged. Returns the number of bytes actually read or -1 if <i>start</i> is beyond the end of the file.
abstract int write(ByteBuffer <i>bb</i>) throws IOException	Writes the contents of <i>bb</i> to the invoking channel, starting at the current position. Returns the number of bytes written.
abstract int write(ByteBuffer <i>bb</i> , long <i>start</i>) throws IOException	Beginning at the file location specified by <i>start</i> , writes the contents of <i>bb</i> to the invoking channel. The current position is unchanged. Returns the number of bytes written.

All channels support additional methods that give you access to and control over the channel. For example, **FileChannel** supports methods to get or set the current position, transfer information between file channels, obtain the current size of the channel, and lock the channel, among others. **FileChannel** provides a **static** method called **open()**, which opens a file and returns a channel to it. This provides another way to obtain a channel. **FileChannel** also provides the **map()** method, which lets you map a file to a buffer.

Charsets and Selectors

Two other entities used by NIO are charsets and selectors. A *charset* defines the way that bytes are mapped to characters. You can encode a sequence of characters into bytes using an *encoder*. You can decode a sequence of bytes into characters using a *decoder*. Charsets, encoders, and decoders are supported by classes defined in the **java.nio.charset** package. Because default encoders and decoders are provided, you will not often need to work explicitly with charsets.

A *selector* supports key-based, non-blocking, multiplexed I/O. In other words, selectors enable you to perform I/O through multiple channels. Selectors are supported by classes defined in the **java.nio.channels** package. Selectors are most applicable to socket-backed channels.

We will not use charsets or selectors in this chapter, but you might find them useful in your own applications.

Enhancements Added to NIO by JDK 7

Beginning with JDK 7, the NIO system was substantially expanded and enhanced. In addition to support for the `try-with-resources` statement (which provides automatic resource management), the improvements included three new packages (`java.nio.file`, `java.nio.file.attribute`, and `java.nio.file.spi`); several new classes, interfaces, and methods; and direct support for stream-based I/O. The additions have greatly expanded the ways in which NIO can be used, especially with files. Several of the key additions are described in the following sections.

The Path Interface

Perhaps the single most important addition to the NIO system is the **Path** interface because it encapsulates a path to a file. As you will see, **Path** is the glue that binds together many of the NIO.2 file-based features. It describes a file's location within the directory structure. **Path** is packaged in `java.nio.file`, and it inherits the following interfaces: `Watchable`, `Iterable<Path>`, and `Comparable<Path>`. `Watchable` describes an object that can be monitored for changes. The `Iterable` and `Comparable` interfaces were described earlier in this book.

Path declares a number of methods that operate on the path. A sampling is shown in Table 21-3. Pay special attention to the `getName()` method. It is used to obtain an element in a path. It works using an index. At index zero is the part of the path nearest the root, which is the leftmost element in a path. Subsequent indexes specify elements to the right of the root. The number of elements in a path can be obtained by calling `getNameCount()`. If you want to obtain a string representation of the entire path, simply call `toString()`. Notice that you can resolve a relative path into an absolute path by using the `resolve()` method.

Method	Description
<code>boolean endsWith(String path)</code>	Returns <code>true</code> if the invoking Path ends with the path specified by <i>path</i> . Otherwise, returns <code>false</code> .
<code>boolean endsWith(Path path)</code>	Returns <code>true</code> if the invoking Path ends with the path specified by <i>path</i> . Otherwise, returns <code>false</code> .
<code>Path getFileName()</code>	Returns the filename associated with the invoking Path .
<code>Path getName(int idx)</code>	Returns a Path object that contains the name of the path element specified by <i>idx</i> within the invoking object. The leftmost element is at index 0. This is the element nearest the root. The rightmost element is at <code>getNameCount() - 1</code> .
<code>int getNameCount()</code>	Returns the number of elements beyond the root directory in the invoking Path .
<code>Path getParent()</code>	Returns a Path that contains the entire path except for the name of the file specified by the invoking Path .
<code>Path getRoot()</code>	Returns the root of the invoking Path .

Table 21-3 A Sampling of Methods Specified by **Path**

Method	Description
boolean isAbsolute()	Returns true if the invoking Path is absolute. Otherwise, returns false .
Path resolve(Path <i>path</i>)	If <i>path</i> is absolute, <i>path</i> is returned. Otherwise, if <i>path</i> does not contain a root, <i>path</i> is prefixed by the root specified by the invoking Path and the result is returned. If <i>path</i> is empty, the invoking Path is returned. Otherwise, the behavior is unspecified.
Path resolve(String <i>path</i>)	If <i>path</i> is absolute, <i>path</i> is returned. Otherwise, if <i>path</i> does not contain a root, <i>path</i> is prefixed by the root specified by the invoking Path and the result is returned. If <i>path</i> is empty, the invoking Path is returned. Otherwise, the behavior is unspecified.
boolean startsWith(String <i>path</i>)	Returns true if the invoking Path starts with the path specified by <i>path</i> . Otherwise, returns false .
boolean startsWith(Path <i>path</i>)	Returns true if the invoking Path starts with the path specified by <i>path</i> . Otherwise, returns false .
Path toAbsolutePath()	Returns the invoking Path as an absolute path.
String toString()	Returns a string representation of the invoking Path .

Table 21-3 A Sampling of Methods Specified by **Path** (*continued*)

One other point: When updating legacy code that uses the **File** class defined by **java.io**, it is possible to convert a **File** instance into a **Path** instance by calling **toPath()** on the **File** object. Furthermore, it is possible to obtain a **File** instance by calling the **toFile()** method defined by **Path**.

The **Files** Class

Many of the actions that you perform on a file are provided by **static** methods within the **Files** class. The file to be acted upon is specified by its **Path**. Thus, the **Files** methods use a **Path** to specify the file that is being operated upon. **Files** contains a wide array of functionality. For example, it has methods that let you open or create a file that has the specified path. You can obtain information about a **Path**, such as whether it is executable, hidden, or read-only. **Files** also supplies methods that let you copy or move files. A sampling is shown in Table 21-4. In addition to **IOException**, several other exceptions are possible. JDK 8 adds these four methods to **Files**: **list()**, **walk()**, **lines()**, and **find()**. All return a **Stream** object. These methods help integrate NIO with the new stream API defined by JDK 8 and described in Chapter 29.

Notice that several of the methods in Table 21-4 take an argument of type **OpenOption**. This is an interface that describes how to open a file. It is implemented by the **StandardOpenOption** class, which defines an enumeration that has the values shown in Table 21-5.

Method	Description
static Path copy(Path <i>src</i> , Path <i>dest</i> , CopyOption ... <i>how</i>) throws IOException	Copies the file specified by <i>src</i> to the location specified by <i>dest</i> . The <i>how</i> parameter specifies how the copy will take place.
static Path createDirectory(Path <i>path</i> , FileAttribute<?> ... <i>attribs</i>) throws IOException	Creates the directory whose path is specified by <i>path</i> . The directory attributes are specified by <i>attribs</i> .
static Path createFile(Path <i>path</i> , FileAttribute<?> ... <i>attribs</i>) throws IOException	Creates the file whose path is specified by <i>path</i> . The file attributes are specified by <i>attribs</i> .
static void delete(Path <i>path</i>) throws IOException	Deletes the file whose path is specified by <i>path</i> .
static boolean exists(Path <i>path</i> , LinkOption ... <i>opts</i>)	Returns true if the file specified by <i>path</i> exists and false otherwise. If <i>opts</i> is not specified, then symbolic links are followed. To prevent the following of symbolic links, pass NOFOLLOW_LINKS to <i>opts</i> .
static boolean isDirectory(Path <i>path</i> , LinkOption ... <i>opts</i>)	Returns true if <i>path</i> specifies a directory and false otherwise. If <i>opts</i> is not specified, then symbolic links are followed. To prevent the following of symbolic links, pass NOFOLLOW_LINKS to <i>opts</i> .
static boolean isExecutable(Path <i>path</i>)	Returns true if the file specified by <i>path</i> is executable and false otherwise.
static boolean isHidden(Path <i>path</i>) throws IOException	Returns true if the file specified by <i>path</i> is hidden and false otherwise.
static boolean isReadable(Path <i>path</i>)	Returns true if the file specified by <i>path</i> can be read from and false otherwise.
static boolean isRegularFile(Path <i>path</i> , LinkOption ... <i>opts</i>)	Returns true if <i>path</i> specifies a file and false otherwise. If <i>opts</i> is not specified, then symbolic links are followed. To prevent the following of symbolic links, pass NOFOLLOW_LINKS to <i>opts</i> .
static boolean isWritable(Path <i>path</i>)	Returns true if the file specified by <i>path</i> can be written to and false otherwise.
static Path move(Path <i>src</i> , Path <i>dest</i> , CopyOption ... <i>how</i>) throws IOException	Moves the file specified by <i>src</i> to the location specified by <i>dest</i> . The <i>how</i> parameter specifies how the move will take place.
static SeekableByteChannel newByteChannel(Path <i>path</i> , OpenOption ... <i>how</i>) throws IOException	Opens the file specified by <i>path</i> , as specified by <i>how</i> . Returns a SeekableByteChannel to the file. This is a byte channel whose current position can be changed. SeekableByteChannel is implemented by FileChannel .
static DirectoryStream<Path> newDirectoryStream(Path <i>path</i>) throws IOException	Opens the directory specified by <i>path</i> . Returns a DirectoryStream linked to the directory.

Table 21-4 A Sampling of Methods Defined by **Files**

Method	Description
static InputStream newInputStream(Path <i>path</i> , OpenOption ... <i>how</i>) throws IOException	Opens the file specified by <i>path</i> , as specified by <i>how</i> . Returns an InputStream linked to the file.
static OutputStream newOutputStream(Path <i>path</i> , OpenOption ... <i>how</i>) throws IOException	Opens the file specified by the invoking object, as specified by <i>how</i> . Returns an OutputStream linked to the file.
static boolean notExists(Path <i>path</i> , LinkOption ... <i>opts</i>)	Returns true if the file specified by <i>path</i> does <i>not</i> exist and false otherwise. If <i>opts</i> is not specified, then symbolic links are followed. To prevent the following of symbolic links, pass NOFOLLOW_LINKS to <i>opts</i> .
static <A extends BasicFileAttributes> A readAttributes(Path <i>path</i> , Class<A> <i>attribType</i> , LinkOption ... <i>opts</i>) throws IOException	Obtains the attributes associated with the file specified by <i>path</i> . The type of attributes to obtain is passed in <i>attribType</i> . If <i>opts</i> is not specified, then symbolic links are followed. To prevent the following of symbolic links, pass NOFOLLOW_LINKS to <i>opts</i> .
static long size(Path <i>path</i>) throws IOException	Returns the size of the file specified by <i>path</i> .

Table 21-4 A Sampling of Methods Defined by **Files** (continued)

Value	Meaning
APPEND	Causes output to be written to the end of the file.
CREATE	Creates the file if it does not already exist.
CREATE_NEW	Creates the file only if it does not already exist.
DELETE_ON_CLOSE	Deletes the file when it is closed.
DSYNC	Causes changes to the file to be immediately written to the physical file. Normally, changes to a file are buffered by the file system in the interest of efficiency, being written to the file only as needed.
READ	Opens the file for input operations.
SPARSE	Indicates to the file system that the file is sparse, meaning that it may not be completely filled with data. If the file system does not support sparse files, this option is ignored.
SYNC	Causes changes to the file or its metadata to be immediately written to the physical file. Normally, changes to a file are buffered by the file system in the interest of efficiency, being written to the file only as needed.
TRUNCATE_EXISTING	Causes a preexisting file opened for output to be reduced to zero length.
WRITE	Opens the file for output operations.

Table 21-5 The Standard Open Options

The Paths Class

Because **Path** is an interface, not a class, you can't create an instance of **Path** directly through the use of a constructor. Instead, you obtain a **Path** by calling a method that returns one. Frequently, you do this by using the **get()** method defined by the **Paths** class. There are two forms of **get()**. The one used in this chapter is shown here:

```
static Path get(String pathname, String ... parts)
```

It returns a **Path** that encapsulates the specified path. The path can be specified in two ways. First, if *parts* is not used, then the path must be specified in its entirety by *pathname*. Alternatively, you can pass the path in pieces, with the first part passed in *pathname* and the subsequent elements specified by the *parts* varargs parameter. In either case, if the path specified is syntactically invalid, **get()** will throw an **InvalidPathException**.

The second form of **get()** creates a **Path** from a **URI**. It is shown here:

```
static Path get(URI uri)
```

The **Path** corresponding to *uri* is returned.

It is important to understand that creating a **Path** to a file does not open or create a file. It simply creates an object that encapsulates the file's directory path.

The File Attribute Interfaces

Associated with a file is a set of attributes. These attributes include such things as the file's time of creation, the time of its last modification, whether the file is a directory, and its size. NIO organizes file attributes into several different interfaces. Attributes are represented by a hierarchy of interfaces defined in **java.nio.file.attribute**. At the top is **BasicFileAttributes**. It encapsulates the set of attributes that are commonly found in a variety of file systems.

The methods defined by **BasicFileAttributes** are shown in Table 21-6.

Method	Description
FileTime creationTime()	Returns the time at which the file was created. If creation time is not provided by the file system, then an implementation-dependent value is returned.
Object fileKey()	Returns the file key. If not supported, null is returned.
boolean isDirectory()	Returns true if the file represents a directory.
boolean isOther()	Returns true if the file is not a file, symbolic link, or a directory.
boolean isRegularFile()	Returns true if the file is a normal file, rather than a directory or symbolic link.
boolean isSymbolicLink()	Returns true if the file is a symbolic link.
FileTime lastAccessTime()	Returns the time at which the file was last accessed. If the time of last access is not provided by the file system, then an implementation-dependent value is returned.
FileTime lastModifiedTime()	Returns the time at which the file was last modified. If the time of last modification is not provided by the file system, then an implementation-dependent value is returned.
long size()	Returns the size of the file.

Table 21-6 The Methods Defined by **BasicFileAttributes**

From **BasicFileAttributes** two interfaces are derived: **DosFileAttributes** and **PosixFileAttributes**. **DosFileAttributes** describes those attributes related to the FAT file system as first defined by DOS. It defines the methods shown here:

Method	Description
boolean isArchive()	Returns true if the file is flagged for archiving and false otherwise.
boolean isHidden()	Returns true if the file is hidden and false otherwise.
boolean isReadOnly()	Returns true if the file is read-only and false otherwise.
boolean isSystem()	Returns true if the file is flagged as a system file and false otherwise.

PosixFileAttributes encapsulates attributes defined by the POSIX standards. (POSIX stands for *Portable Operating System Interface*.) It defines the methods shown here:

Method	Description
GroupPrincipal group()	Returns the file's group owner.
UserPrincipal owner()	Returns the file's owner.
Set<PosixFilePermission> permissions()	Returns the file's permissions.

There are various ways to access a file's attributes. First, you can obtain an object that encapsulates a file's attributes by calling **readAttributes()**, which is a **static** method defined by **Files**. One of its forms is shown here:

```
static <A extends BasicFileAttributes>
    A readAttributes(Path path, Class<A> attrType, LinkOption... opts)
        throws IOException
```

This method returns a reference to an object that specifies the attributes associated with the file passed in *path*. The specific type of attributes is specified as a **Class** object in the *attrType* parameter. For example, to obtain the basic file attributes, pass **BasicFileAttributes.class** to *attrType*. For DOS attributes, use **DosFileAttributes.class**, and for POSIX attributes, use **PosixFileAttributes.class**. Optional link options are passed via *opts*. If not specified, symbolic links are followed. The method returns a reference to requested attributes. If the requested attribute type is not available, **UnsupportedOperationException** is thrown. Using the object returned, you can access the file's attributes.

A second way to gain access to a file's attributes is to call **getFileAttributeView()** defined by **Files**. NIO defines several attribute view interfaces, including **AttributeView**, **BasicFileAttributeView**, **DosFileAttributeView**, and **PosixFileAttributeView**, among others. Although we won't be using attribute views in this chapter, they are a feature that you may find helpful in some situations.

In some cases, you won't need to use the file attribute interfaces directly because the **Files** class offers **static** convenience methods that access several of the attributes. For example, **Files** includes methods such as **isHidden()** and **isWritable()**.

It is important to understand that not all file systems support all possible attributes. For example, the DOS file attributes apply to the older FAT file system as first defined by DOS. The attributes that will apply to a wide variety of file systems are described by **BasicFileAttributes**. For this reason, these attributes are used in the examples in this chapter.

The FileSystem, FileSystems, and FileStore Classes

You can easily access the file system through the **FileSystem** and **FileSystems** classes packaged in **java.nio.file**. In fact, by using the **newFileSystem()** method defined by **FileSystems**, it is even possible to obtain a new file system. The **FileStore** class encapsulates the file storage system. Although these classes are not used directly in this chapter, you may find them helpful in your own applications.

Using the NIO System

This section illustrates how to apply the NIO system to a variety of tasks. Before beginning, it is important to emphasize that with the release of JDK 7, the NIO subsystem was greatly expanded. As a result, its uses have also been greatly expanded. As mentioned, the enhanced version is sometimes referred to as NIO.2. Because the features added by NIO.2 are so substantial, they have changed the way that much NIO-based code is written and have increased the types of tasks to which NIO can be applied. Because of its importance, most of the remaining discussion and examples in this chapter utilize NIO.2 features and, therefore, require JDK 7, JDK 8, or later. However, at the end of the chapter is a brief description of pre-JDK 7 code. This will be of aid to those programmers working in pre-JDK 7 environments or maintaining older code.

REMEMBER Most of the examples in this chapter require JDK 7 or later.

In the past, the primary purpose of NIO was channel-based I/O, and this is still a very important use. However, you can now use NIO for stream-based I/O and for performing file-system operations. As a result, the discussion of using NIO is divided into three parts:

- Using NIO for channel-based I/O
- Using NIO for stream-based I/O
- Using NIO for path and file system operations

Because the most common I/O device is the disk file, the rest of this chapter uses disk files in the examples. Because all file channel operations are byte-based, the type of buffers that we will be using are of type **ByteBuffer**.

Before you can open a file for access via the NIO system, you must obtain a **Path** that describes the file. One way to do this is to call the **Paths.get()** factory method, which was described earlier. The form of **get()** used in the examples is shown here:

```
static Path get(String pathname, String ... parts)
```

Recall that the path can be specified in two ways. It can be passed in pieces, with the first part passed in *pathname* and the subsequent elements specified by the *parts* varargs parameter. Alternatively, the entire path can be specified in *pathname* and *parts* is not used. This is the approach used by the examples.

Use NIO for Channel-Based I/O

An important use of NIO is to access a file via a channel and buffers. The following sections demonstrate some techniques that use a channel to read from and write to a file.

Reading a File via a Channel

There are several ways to read data from a file using a channel. Perhaps the most common way is to manually allocate a buffer and then perform an explicit read operation that loads that buffer with data from the file. It is with this approach that we begin.

Before you can read from a file, you must open it. To do this, first create a **Path** that describes the file. Then use this **Path** to open the file. There are various ways to open the file depending on how it will be used. In this example, the file will be opened for byte-based input via explicit input operations. Therefore, this example will open the file and establish a channel to it by calling **Files.newByteChannel()**. The **newByteChannel()** method has this general form:

```
static SeekableByteChannel newByteChannel(Path path, OpenOption ... how)
    throws IOException
```

It returns a **SeekableByteChannel** object, which encapsulates the channel for file operations. The **Path** that describes the file is passed in *path*. The *how* parameter specifies how the file will be opened. Because it is a varargs parameter, you can specify zero or more comma-separated arguments. (The valid values were discussed earlier and shown in Table 21-5.) If no arguments are specified, the file is opened for input operations. **SeekableByteChannel** is an interface that describes a channel that can be used for file operations. It is implemented by the **FileChannel** class. When the default file system is used, the returned object can be cast to **FileChannel**. You must close the channel after you have finished with it. Since all channels, including **FileChannel**, implement **AutoCloseable**, you can use a **try-with-resources** statement to close the file automatically instead of calling **close()** explicitly. This approach is used in the examples.

Next, you must obtain a buffer that will be used by the channel either by wrapping an existing array or by allocating the buffer dynamically. The examples use allocation, but the choice is yours. Because file channels operate on byte buffers, we will use the **allocate()** method defined by **ByteBuffer** to obtain the buffer. It has this general form:

```
static ByteBuffer allocate(int cap)
```

Here, *cap* specifies the capacity of the buffer. A reference to the buffer is returned.

After you have created the buffer, call **read()** on the channel, passing a reference to the buffer. The version of **read()** that we will use is shown next:

```
int read(ByteBuffer buf) throws IOException
```

Each time it is called, **read()** fills the buffer specified by *buf* with data from the file. The reads are sequential, meaning that each call to **read()** reads the next buffer's worth of bytes from the file. The **read()** method returns the number of bytes actually read. It returns **-1** when there is an attempt to read at the end of the file.

The following program puts the preceding discussion into action by reading a file called **test.txt** through a channel using explicit input operations:

```
// Use Channel I/O to read a file. Requires JDK 7 or later.

import java.io.*;
import java.nio.*;
import java.nio.channels.*;
import java.nio.file.*;
```

```

public class ExplicitChannelRead {
    public static void main(String args[]) {
        int count;
        Path filepath = null;

        // First, obtain a path to the file.
        try {
            filepath = Paths.get("test.txt");
        } catch(InvalidPathException e) {
            System.out.println("Path Error " + e);
            return;
        }

        // Next, obtain a channel to that file within a try-with-resources block.
        try ( SeekableByteChannel fChan = Files.newByteChannel(filepath) )
        {

            // Allocate a buffer.
            ByteBuffer mBuf = ByteBuffer.allocate(128);

            do {
                // Read a buffer.
                count = fChan.read(mBuf);

                // Stop when end of file is reached.
                if(count != -1) {

                    // Rewind the buffer so that it can be read.
                    mBuf.rewind();

                    // Read bytes from the buffer and show
                    // them on the screen as characters.
                    for(int i=0; i < count; i++)
                        System.out.print((char)mBuf.get());
                }
            } while(count != -1);

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

```

Here is how the program works. First, a **Path** object is obtained that contains the relative path to a file called **test.txt**. A reference to this object is assigned to **filepath**. Next, a channel connected to the file is obtained by calling **newByteChannel()**, passing in **filepath**. Because no open option is specified, the file is opened for reading. Notice that this channel is the object managed by the **try-with-resources** statement. Thus, the channel is automatically closed when the block ends. The program then calls the **allocate()** method of **ByteBuffer** to allocate a buffer that will hold the contents of the file when it is read. A reference to this buffer is stored in **mBuf**. The contents of the file are then read, one buffer at a time, into **mBuf** through a call to **read()**. The number of bytes read is stored in **count**. Next, the

buffer is rewound through a call to `rewind()`. This call is necessary because the current position is at the end of the buffer after the call to `read()`. It must be reset to the start of the buffer in order for the bytes in `mBuf` to be read by calling `get()`. (Recall that `get()` is defined by `ByteBuffer`.) Because `mBuf` is a byte buffer, the values returned by `get()` are bytes. They are cast to `char` so the file can be displayed as text. (Alternatively, it is possible to create a buffer that encodes the bytes into characters and then read that buffer.) When the end of the file has been reached, the value returned by `read()` will be `-1`. When this occurs, the program ends, and the channel is automatically closed.

As a point of interest, notice that the program obtains the `Path` within one `try` block and then uses another `try` block to obtain and manage a channel linked to that path. Although there is nothing wrong, per se, with this approach, in many cases, it can be streamlined so that only one `try` block is needed. In this approach, the calls to `Paths.get()` and `newByteChannel()` are sequenced together. For example, here is a reworked version of the program that uses this approach:

```
// A more compact way to open a channel. Requires JDK 7 or later.

import java.io.*;
import java.nio.*;
import java.nio.channels.*;
import java.nio.file.*;

public class ExplicitChannelRead {
    public static void main(String args[]) {
        int count;

        // Here, the channel is opened on the Path returned by Paths.get().
        // There is no need for the filepath variable.
        try ( SeekableByteChannel fChan =
                  Files.newByteChannel(Paths.get("test.txt")) )
        {
            // Allocate a buffer.
            ByteBuffer mBuf = ByteBuffer.allocate(128);

            do {
                // Read a buffer.
                count = fChan.read(mBuf);

                // Stop when end of file is reached.
                if(count != -1) {

                    // Rewind the buffer so that it can be read.
                    mBuf.rewind();

                    // Read bytes from the buffer and show
                    // them on the screen as characters.
                    for(int i=0; i < count; i++)
                        System.out.print((char)mBuf.get());
                }
            } while(count != -1);
        }
    }
}
```

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

In this version, the variable **filepath** is not needed and both exceptions are handled by the same **try** statement. Because this approach is more compact, it is the approach used in the rest of the examples in this chapter. Of course, in your own code, you may encounter situations in which the creation of a **Path** object needs to be separate from the acquisition of a channel. In these cases, the previous approach can be used.

Another way to read a file is to map it to a buffer. The advantage is that the buffer automatically contains the contents of the file. No explicit read operation is necessary. To map and read the contents of a file, follow this general procedure. First, obtain a **Path** object that encapsulates the file as previously described. Next, obtain a channel to that file by calling **Files.newByteChannel()**, passing in the **Path** and casting the returned object to **FileChannel**. As explained, **newByteChannel()** returns a **SeekableByteChannel**. When using the default file system, this object can be cast to **FileChannel**. Then, map the channel to a buffer by calling **map()** on the channel. The **map()** method is defined by **FileChannel**. This is why the cast to **FileChannel** is needed. The **map()** function is shown here:

```
MappedByteBuffer map(FileChannel.MapMode how,  
                    long pos, long size) throws IOException
```

The `map()` method causes the data in the file to be mapped into a buffer in memory. The value in `how` determines what type of operations are allowed. It must be one of these values:

MapMode.READ_ONLY MapMode.READ_WRITE MapMode.PRIVATE

For reading a file, use **MapMode.READ_ONLY**. To read and write, use **MapMode.READ_WRITE**. **MapMode.PRIVATE** causes a private copy of the file to be made, and changes to the buffer do not affect the underlying file. The location within the file to begin mapping is specified by *pos*, and the number of bytes to map are specified by *size*. A reference to this buffer is returned as a **MappedByteBuffer**, which is a subclass of **ByteBuffer**. Once the file has been mapped to a buffer, you can read the file from that buffer. Here is an example that illustrates this approach:

// Use a mapped file to read a file. Requires JDK 7 or later.

```
import java.io.*;
import java.nio.*;
import java.nio.channels.*;
import java.nio.file.*;

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

```
// Obtain a channel to a file within a try-with-resources block.
try ( FileChannel fChan =
      (FileChannel) Files.newByteChannel(Paths.get("test.txt")) ) {
    // Get the size of the file.
    long fSize = fChan.size();

    // Now, map the file into a buffer.
    MappedByteBuffer mBuf = fChan.map(FileChannel.MapMode.READ_ONLY, 0, fSize);

    // Read and display bytes from buffer.
    for(int i=0; i < fSize; i++)
        System.out.print((char)mBuf.get());

    System.out.println();
}

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

In the program, a **Path** to the file is created and then opened via **newByteChannel()**. The channel is cast to **FileChannel** and stored in **fChan**. Next, the size of the file is obtained by calling **size()** on the channel. Then, the entire file is mapped into memory by calling **map()** on **fChan** and a reference to the buffer is stored in **mBuf**. Notice that **mBuf** is declared as a reference to a **MappedByteBuffer**. The bytes in **mBuf** are read by calling **get()**.

Writing to a File via a Channel

As is the case when reading from a file, there are also several ways to write data to a file using a channel. We will begin with one of the most common. In this approach, you manually allocate a buffer, write data to that buffer, and then perform an explicit write operation to write that data to a file.

Before you can write to a file, you must open it. To do this, first obtain a **Path** that describes the file and then use this **Path** to open the file. In this example, the file will be opened for byte-based output via explicit output operations. Therefore, this example will open the file and establish a channel to it by calling **Files.newByteChannel()**. As shown in the previous section, the **newByteChannel()** method has this general form:

```
static SeekableByteChannel newByteChannel(Path path, OpenOption ... how)  
    throws IOException
```

It returns a **SeekableByteChannel** object, which encapsulates the channel for file operations. To open a file for output, the *how* parameter must specify **StandardOpenOption.WRITE**. If you want to create the file if it does not already exist, then you must also specify **StandardOpenOption.CREATE**. (Other options, which are shown in Table 21-5, are also available.) As explained in the previous section, **SeekableByteChannel** is an interface that describes a channel that can be used for file operations. It is implemented by the **FileChannel**

class. When the default file system is used, the return object can be cast to **FileChannel**. You must close the channel after you have finished with it.

Here is one way to write to a file through a channel using explicit calls to **write()**. First, obtain a **Path** to the file and then open it with a call to **newByteChannel()**, casting the result to **FileChannel**. Next, allocate a byte buffer and write data to that buffer. Before the data is written to the file, call **rewind()** on the buffer to set its current position to zero. (Each output operation on the buffer increases the current position. Thus, it must be reset prior to writing to the file.) Then, call **write()** on the channel, passing in the buffer. The following program demonstrates this procedure. It writes the alphabet to a file called **test.txt**.

```
// Write to a file using NIO. Requires JDK 7 or later.

import java.io.*;
import java.nio.*;
import java.nio.channels.*;
import java.nio.file.*;

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

        // Obtain a channel to a file within a try-with-resources block.
        try ( FileChannel fChan = (FileChannel)
              Files.newByteChannel(Paths.get("test.txt"),
                                   StandardOpenOption.WRITE,
                                   StandardOpenOption.CREATE) )
        {
            // Create a buffer.
            ByteBuffer mBuf = ByteBuffer.allocate(26);

            // Write some bytes to the buffer.
            for(int i=0; i<26; i++)
                mBuf.put((byte)('A' + i));

            // Reset the buffer so that it can be written.
            mBuf.rewind();

            // Write the buffer to the output file.
            fChan.write(mBuf);

        } catch(InvalidPathException e) {
            System.out.println("Path Error " + e);
        } catch (IOException e) {
            System.out.println("I/O Error: " + e);
            System.exit(1);
        }
    }
}
```

It is useful to emphasize an important aspect of this program. As mentioned, after data is written to **mBuf**, but before it is written to the file, a call to **rewind()** on **mBuf** is made. This is necessary in order to reset the current position to zero after data has been written to **mBuf**. Remember, each call to **put()** on **mBuf** advances the current position. Therefore,

it is necessary for the current position to be reset to the start of the buffer before calling `write()`. If this is not done, `write()` will think that there is no data in the buffer.

Another way to handle the resetting of the buffer between input and output operations is to call `flip()` instead of `rewind()`. The `flip()` method sets the value of the current position to zero and the limit to the previous current position. In the preceding example, because the capacity of the buffer equals its limit, `flip()` could have been used instead of `rewind()`. However, the two methods are not interchangeable in all cases.

In general, you must reset the buffer between read and write operations. For example, assuming the preceding example, the following loop will write the alphabet to the file three times. Pay special attention to the calls to `rewind()`.

```
for(int h=0; h<3; h++) {
    // Write some bytes to the buffer.
    for(int i=0; i<26; i++)
        mBuf.put((byte) ('A' + i));

    // Rewind the buffer so that it can be written.
    mBuf.rewind();

    // Write the buffer to the output file.
    fChan.write(mBuf);

    // Rewind the buffer so that it can be written to again.
    mBuf.rewind();
}
```

Notice that `rewind()` is called between each read and write operation.

One other thing about the program warrants mentioning: When the buffer is written to the file, the first 26 bytes in the file will contain the output. If the file `test.txt` was preexisting, then after the program executes, the first 26 bytes of `test.txt` will contain the alphabet, but the remainder of the file will remain unchanged.

Another way to write to a file is to map it to a buffer. The advantage to this approach is that the data written to the buffer will automatically be written to the file. No explicit write operation is necessary. To map and write the contents of a file, we will use this general procedure. First, obtain a `Path` object that encapsulates the file and then create a channel to that file by calling `Files.newByteChannel()`, passing in the `Path`. Cast the reference returned by `newByteChannel()` to `FileChannel`. Next, map the channel to a buffer by calling `map()` on the channel. The `map()` method was described in detail in the previous section. It is summarized here for your convenience. Here is its general form:

```
MappedByteBuffer map(FileChannel.MapMode how,
                     long pos, long size) throws IOException
```

The `map()` method causes the data in the file to be mapped into a buffer in memory. The value in `how` determines what type of operations are allowed. For writing to a file, `how` must be `MapMode.READ_WRITE`. The location within the file to begin mapping is specified by `pos`, and the number of bytes to map are specified by `size`. A reference to this buffer is returned. Once the file has been mapped to a buffer, you can write data to that buffer, and it will automatically be written to the file. Therefore, no explicit write operations to the channel are necessary.

Here is the preceding program reworked so that a mapped file is used. Notice that in the call to `newByteChannel()`, the open option `StandardOpenOption.READ` has been added. This is because a mapped buffer can either be read-only or read/write. Thus, to write to the mapped buffer, the channel must be opened as read/write.

```
// Write to a mapped file. Requires JDK 7 or later.

import java.io.*;
import java.nio.*;
import java.nio.channels.*;
import java.nio.file.*;

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

        // Obtain a channel to a file within a try-with-resources block.
        try ( FileChannel fChan = (FileChannel)
              Files.newByteChannel(Paths.get("test.txt"),
                                   StandardOpenOption.WRITE,
                                   StandardOpenOption.READ,
                                   StandardOpenOption.CREATE) )
        {

            // Then, map the file into a buffer.
            MappedByteBuffer mBuf = fChan.map(FileChannel.MapMode.READ_WRITE, 0, 26);

            // Write some bytes to the buffer.
            for(int i=0; i<26; i++)
                mBuf.put((byte)('A' + i));

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

As you can see, there are no explicit write operations to the channel itself. Because `mBuf` is mapped to the file, changes to `mBuf` are automatically reflected in the underlying file.

Copying a File Using NIO

NIO simplifies several types of file operations. Although we can't examine them all, an example will give you an idea of what is available. The following program copies a file using a call to a single NIO method: `copy()`, which is a `static` method defined by `Files`. It has several forms. Here is the one we will be using:

```
static Path copy(Path src, Path dest, CopyOption ... how) throws IOException
```

The file specified by `src` is copied to the file specified by `dest`. How the copy is performed is specified by `how`. Because it is a varargs parameter, it can be missing. If specified, it can be one or more of these values, which are valid for all file systems:

StandardCopyOption.COPY_ATTRIBUTES	Request that the file's attributes be copied.
StandardLinkOption.NOFOLLOW_LINKS	Do not follow symbolic links.
StandardCopyOption.REPLACE_EXISTING	Overwrite a preexisting file.

Other options may be supported, depending on the implementation.

The following program demonstrates `copy()`. The source and destination files are specified on the command line, with the source file specified first. Notice how short the program is. You might want to compare this version of the file copy program to the one found in Chapter 13. As you will find, the part of the program that actually copies the file is substantially shorter in the NIO version shown here.

```
// Copy a file using NIO. Requires JDK 7 or later.
import java.io.*;
import java.nio.*;
import java.nio.channels.*;
import java.nio.file.*;

public class NIOCopy {

    public static void main(String args[]) {

        if(args.length != 2) {
            System.out.println("Usage: Copy from to");
            return;
        }

        try {
            Path source = Paths.get(args[0]);
            Path target = Paths.get(args[1]);

            // Copy the file.
            Files.copy(source, target, StandardCopyOption.REPLACE_EXISTING);

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

Use NIO for Stream-Based I/O

Beginning with NIO.2, you can use NIO to open an I/O stream. Once you have a **Path**, open a file by calling `newInputStream()` or `newOutputStream()`, which are **static** methods defined by **Files**. These methods return a stream connected to the specified file. In either case, the stream can then be operated on in the way described in Chapter 20, and the same techniques apply. The advantage of using **Path** to open a file is that all of the features defined by NIO are available for your use.

To open a file for stream-based input, use **Files.newInputStream()**. It is shown here:

```
static InputStream newInputStream(Path path, OpenOption ... how)
    throws IOException
```

Here, *path* specifies the file to open and *how* specifies how the file will be opened. It must be one or more of the values defined by **StandardOpenOption**, described earlier. (Of course, only those options that relate to an input stream will apply.) If no options are specified, then the file is opened as if **StandardOpenOption.READ** were passed.

Once opened, you can use any of the methods defined by **InputStream**. For example, you can use **read()** to read bytes from the file.

The following program demonstrates the use of NIO-based stream I/O. It reworks the **ShowFile** program from Chapter 13 so that it uses NIO features to open the file and obtain a stream. As you can see, it is very similar to the original, except for the use of **Path** and **newInputStream()**.

```
/* Display a text file using stream-based, NIO code.
Requires JDK 7 or later.

To use this program, specify the name
of the file that you want to see.
For example, to see a file called TEST.TXT,
use the following command line.

java ShowFile TEST.TXT
 */

import java.io.*;
import java.nio.file.*;

class ShowFile {
    public static void main(String args[])
    {
        int i;

        // First, confirm that a filename has been specified.
        if(args.length != 1) {
            System.out.println("Usage: ShowFile filename");
            return;
        }

        // Open the file and obtain a stream linked to it.
        try ( InputStream fin = Files.newInputStream(Paths.get(args[0])) )
        {
            do {
                i = fin.read();
                if(i != -1) System.out.print((char) i);
            } while(i != -1);

        } catch(InvalidPathException e) {
            System.out.println("Path Error " + e);
        } catch(IOException e) {
```

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

Because the stream returned by `newInputStream()` is a normal stream, it can be used like any other stream. For example, you can wrap the stream inside a buffered stream, such as a `BufferedInputStream`, to provide buffering, as shown here:

```
new BufferedInputStream(Files.newInputStream(Paths.get(args[0]))))
```

Now, all reads will be automatically buffered.

To open a file for output, use **Files.newOutputStream()**. It is shown here:

```
static OutputStream newOutputStream(Path path, OpenOption ... how)  
    throws IOException
```

Here, *path* specifies the file to open and *how* specifies how the file will be opened. It must be one or more of the values defined by **StandardOpenOption**, described earlier. (Of course, only those options that relate to an output stream will apply.) If no options are specified, then the file is opened as if **StandardOpenOption.WRITE**, **StandardOpenOption.CREATE**, and **StandardOpenOption.TRUNCATE_EXISTING** were passed.

The methodology for using `newOutputStream()` is similar to that shown previously for `newInputStream()`. Once opened, you can use any of the methods defined by `OutputStream`. For example, you can use `write()` to write bytes to the file. You can also wrap the stream inside a `BufferedOutputStream` to buffer the stream.

The following program shows `newOutputStream()` in action. It writes the alphabet to a file called `test.txt`. Notice the use of buffered I/O.

```
// Demonstrate NIO-based, stream output. Requires JDK 7 or later.

import java.io.*;
import java.nio.file.*;

class NIOStreamWrite {
    public static void main(String args[])
    {
        // Open the file and obtain a stream linked to it.
        try ( OutputStream fout =
                new BufferedOutputStream(
                    Files.newOutputStream(Paths.get("test.txt")))) {
            // Write some bytes to the stream.
            for(int i=0; i < 26; i++)
                fout.write((byte)('A' + i));

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

Use NIO for Path and File System Operations

At the beginning of Chapter 20, the **File** class in the **java.io** package was examined. As explained there, the **File** class deals with the file system and with the various attributes associated with a file, such as whether a file is read-only, hidden, and so on. It was also used to obtain information about a file's path. Although the **File** class is still perfectly acceptable, the interfaces and classes defined by NIO.2 offer a better way to perform these functions. The benefits include support for symbolic links, better support for directory tree traversal, and improved handling of metadata, among others. The following sections show samples of two common file system operations: obtaining information about a path and file and getting the contents of a directory.

REMEMBER If you want to update older code that uses **java.io.File** to the new **Path** interface, you can use the **toPath()** method to obtain a **Path** instance from a **File** instance.

Obtain Information About a Path and a File

Information about a path can be obtained by using methods defined by **Path**. Some attributes associated with the file described by a **Path** (such as whether or not the file is hidden) are obtained by using methods defined by **Files**. The **Path** methods used here are **getName()**, **getParent()**, and **toAbsolutePath()**. Those provided by **Files** are **isExecutable()**, **isHidden()**, **isReadable()**, **isWritable()**, and **exists()**. These are summarized in Tables 21-3 and 21-4, shown earlier.

CAUTION Methods such as **isExecutable()**, **isReadable()**, **isWritable()**, and **exists()** must be used with care because the state of the file system may change after the call, in which case a program malfunction could occur. Such a situation could have security implications.

Other file attributes are obtained by requesting a list of attributes by calling **Files.readAttributes()**. In the program, this method is called to obtain the **BasicFileAttributes** associated with a file, but the general approach applies to other types of attributes.

The following program demonstrates several of the **Path** and **Files** methods, along with several methods provided by **BasicFileAttributes**. This program assumes that a file called **test.txt** exists in a directory called **examples**, which must be a subdirectory of the current directory.

```
// Obtain information about a path and a file.
// Requires JDK 7 or later.

import java.io.*;
import java.nio.file.*;
import java.nio.file.attribute.*;

class PathDemo {
    public static void main(String args[]) {
        Path filepath = Paths.get("examples\\test.txt");
        BasicFileAttributes attrs = Files.readAttributes(filepath,
                BasicFileAttributes.class);
        System.out.println(attrs.size());
        System.out.println(attrs.lastModifiedTime());
        System.out.println(attrs.lastAccessTime());
        System.out.println(attrs.isDirectory());
        System.out.println(attrs.isExecutable());
        System.out.println(attrs.isHidden());
        System.out.println(attrs.isReadable());
        System.out.println(attrs.isWritable());
        System.out.println(attrs.exists());
```

```
System.out.println("File Name: " + filepath.getName(1));
System.out.println("Path: " + filepath);
System.out.println("Absolute Path: " + filepath.toAbsolutePath());
System.out.println("Parent: " + filepath.getParent());

if(Files.exists(filepath))
    System.out.println("File exists");
else
    System.out.println("File does not exist");

try {
    if(Files.isHidden(filepath))
        System.out.println("File is hidden");
    else
        System.out.println("File is not hidden");
} catch(IOException e) {
    System.out.println("I/O Error: " + e);
}

Files.isWritable(filepath);
System.out.println("File is writable");

Files.isReadable(filepath);
System.out.println("File is readable");

try {
    BasicFileAttributes attrs =
        Files.readAttributes(filepath, BasicFileAttributes.class);

    if(attrs.isDirectory())
        System.out.println("The file is a directory");
    else
        System.out.println("The file is not a directory");

    if(attrs.isRegularFile())
        System.out.println("The file is a normal file");
    else
        System.out.println("The file is not a normal file");

    if(attrs.isSymbolicLink())
        System.out.println("The file is a symbolic link");
    else
        System.out.println("The file is not a symbolic link");

    System.out.println("File last modified: " + attrs.lastModifiedTime());
    System.out.println("File size: " + attrs.size() + " Bytes");
} catch(IOException e) {
    System.out.println("Error reading attributes: " + e);
}
}
```

If you execute this program from a directory called **MyDir**, which has a subdirectory called **examples**, and the **examples** directory contains the **test.txt** file, then you will see output similar to that shown here. (Of course, the information you see will differ.)

```
File Name: test.txt
Path: examples\test.txt
Absolute Path: C:\MyDir\examples\test.txt
Parent: examples
File exists
File is not hidden
File is writable
File is readable
The file is not a directory
The file is a normal file
The file is not a symbolic link
File last modified: 2014-01-01T18:20:46.380445Z
File size: 18 Bytes
```

If you are using a computer that supports the FAT file system (i.e., the DOS file system), then you might want to try using the methods defined by **DosFileAttributes**. If you are using a POSIX-compatible system, then try using **PosixFileAttributes**.

List the Contents of a Directory

If a path describes a directory, then you can read the contents of that directory by using **static** methods defined by **Files**. To do this, you first obtain a directory stream by calling **newDirectoryStream()**, passing in a **Path** that describes the directory. One form of **newDirectoryStream()** is shown here:

```
static DirectoryStream<Path> newDirectoryStream(Path dirPath)
    throws IOException
```

Here, *dirPath* encapsulates the path to the directory. The method returns a **DirectoryStream<Path>** object that can be used to obtain the contents of the directory. It will throw an **IOException** if an I/O error occurs and a **NotDirectoryException** (which is a subclass of **IOException**) if the specified path is not a directory. A **SecurityException** is also possible if access to the directory is not permitted.

DirectoryStream<Path> implements **AutoCloseable**, so it can be managed by a **try-with-resources** statement. It also implements **Iterable<Path>**. This means that you can obtain the contents of the directory by iterating over the **DirectoryStream** object. When iterating, each directory entry is represented by a **Path** instance. An easy way to iterate over a **DirectoryStream** is to use a for-each style **for** loop. It is important to understand, however, that the iterator implemented by **DirectoryStream<Path>** can be obtained only once for each instance. Thus, the **iterator()** method can be called only once, and a for-each loop can be executed only once.

The following program displays the contents of a directory called **MyDir**:

```
// Display a directory. Requires JDK 7 or later.

import java.io.*;
import java.nio.file.*;
import java.nio.file.attribute.*;
```

```

class DirList {
    public static void main(String args[]) {
        String dirname = "\\\MyDir";

        // Obtain and manage a directory stream within a try block.
        try ( DirectoryStream<Path> dirstrm =
                  Files.newDirectoryStream(Paths.get(dirname)) )
        {
            System.out.println("Directory of " + dirname);

            // Because DirectoryStream implements Iterable, we
            // can use a "foreach" loop to display the directory.
            for(Path entry : dirstrm) {
                BasicFileAttributes attrs =
                    Files.readAttributes(entry, BasicFileAttributes.class);

                if(attrs.isDirectory())
                    System.out.print("<DIR> ");
                else
                    System.out.print("      ");

                System.out.println(entry.getName(1));
            }
        } catch(InvalidPathException e) {
            System.out.println("Path Error " + e);
        } catch(NotDirectoryException e) {
            System.out.println(dirname + " is not a directory.");
        } catch (IOException e) {
            System.out.println("I/O Error: " + e);
        }
    }
}

```

Here is sample output from the program:

```

Directory of \MyDir
DirList.class
DirList.java
<DIR> examples
Test.txt

```

You can filter the contents of a directory in two ways. The easiest is to use this version of `newDirectoryStream()`:

```

static DirectoryStream<Path> newDirectoryStream(Path dirPath, String wildcard)
    throws IOException

```

In this version, only files that match the wildcard filename specified by *wildcard* will be obtained. For *wildcard*, you can specify either a complete filename or a *glob*. A *glob* is a string that defines a general pattern that will match one or more files using the familiar * and ? wildcard characters. These match zero or more of any character and any one character, respectively. The following are also recognized within a glob.

**	Matches zero or more of any character across directories.
[<i>chars</i>]	Matches any one character in <i>chars</i> . A * or ? within <i>chars</i> will be treated as a normal character, not a wildcard. A range can be specified by use of a hyphen, such as [x-z].
{ <i>globlist</i> }	Matches any one of the globs specified in a comma-separated list of globs in <i>globlist</i> .

You can specify a * or ? character, using * and \?. To specify a \, use \\. You can experiment with a glob by substituting this call to **newDirectoryStream()** into the previous program:

```
Files.newDirectoryStream(Paths.get(dirname), "{Path,Dir}*.{java,class}")
```

This obtains a directory stream that contains only those files whose names begin with either "Path" or "Dir" and use either the "java" or "class" extension. Thus, it would match names like **DirList.java** and **PathDemo.java**, but not **MyPathDemo.java**, for example.

Another way to filter a directory is to use this version of **newDirectoryStream()**:

```
static DirectoryStream<Path> newDirectoryStream(Path dirPath,
    DirectoryStream.Filter<? super Path> filefilter)
    throws IOException
```

Here, **DirectoryStream.Filter** is an interface that specifies the following method:

```
boolean accept(T entry) throws IOException
```

In this case, T will be **Path**. If you want to include *entry* in the list, return **true**. Otherwise, return **false**. This form of **newDirectoryStream()** offers the advantage of being able to filter a directory based on something other than a filename. For example, you can filter based on size, creation date, modification date, or attribute, to name a few.

The following program demonstrates the process. It will list only those files that are writable.

```
// Display a directory of only those files that are writable.

import java.io.*;
import java.nio.file.*;
import java.nio.file.attribute.*;

class DirList {
    public static void main(String args[]) {
        String dirname = "\\\\"MyDir";

        // Create a filter that returns true only for writable files.
        DirectoryStream.Filter<Path> how = new DirectoryStream.Filter<Path>() {
            public boolean accept(Path filename) throws IOException {
                if(Files.isWritable(filename)) return true;
                return false;
            }
        };
    }
}
```

```
// Obtain and manage a directory stream of writable files.
try (DirectoryStream<Path> dirstrm =
        Files.newDirectoryStream(Paths.get(dirname), how) )
{
    System.out.println("Directory of " + dirname);

    for(Path entry : dirstrm) {
        BasicFileAttributes attrs =
            Files.readAttributes(entry, BasicFileAttributes.class);

        if(attrs.isDirectory())
            System.out.print("<DIR> ");
        else
            System.out.print("          ");

        System.out.println(entry.getName(1));
    }
} catch(InvalidPathException e) {
    System.out.println("Path Error " + e);
} catch(NotDirectoryException e) {
    System.out.println(dirname + " is not a directory.");
} catch (IOException e) {
    System.out.println("I/O Error: " + e);
}
}
```

Use walkFileTree() to List a Directory Tree

The preceding examples have obtained the contents of only a single directory. However, sometimes you will want to obtain a list of the files in a directory tree. In the past, this was quite a chore, but NIO.2 makes it easy because now you can use the `walkFileTree()` method defined by `Files` to process a directory tree. It has two forms. The one used in this chapter is shown here:

```
static Path walkFileTree(Path root, FileVisitor<? extends Path> fv)
    throws IOException
```

The path to the starting point of the directory walk is passed in `root`. An instance of `FileVisitor` is passed in `fv`. The implementation of `FileVisitor` determines how the directory tree is traversed, and it gives you access to the directory information. If an I/O error occurs, an `IOException` is thrown. A `SecurityException` is also possible.

FileVisitor is an interface that defines how files are visited when a directory tree is traversed. It is a generic interface that is declared like this:

interface FileVisitor<T>

For use in `walkFileTree()`, `T` will be `Path` (or any type derived from `Path`). `FileVisitor` defines the following methods.

Method	Description
<code>FileVisitResult postVisitDirectory(T dir, IOException exc) throws IOException</code>	Called after a directory has been visited. The directory is passed in <code>dir</code> , and any <code>IOException</code> is passed in <code>exc</code> . If <code>exc</code> is <code>null</code> , no exception occurred. The result is returned.
<code>FileVisitResult preVisitDirectory(T dir, BasicFileAttributes attrs) throws IOException</code>	Called before a directory is visited. The directory is passed in <code>dir</code> , and the attributes associated with the directory are passed in <code>attrs</code> . The result is returned. To examine the directory, return <code>FileVisitResult.CONTINUE</code> .
<code>FileVisitResult visitFile(T file, BasicFileAttributes attrs) throws IOException</code>	Called when a file is visited. The file is passed in <code>file</code> , and the attributes associated with the file are passed in <code>attrs</code> . The result is returned.
<code>FileVisitResult visitFileFailed(T file, IOException exc) throws IOException</code>	Called when an attempt to visit a file fails. The file that failed is passed in <code>file</code> , and the <code>IOException</code> is passed in <code>exc</code> . The result is returned.

Notice that each method returns a `FileVisitResult`. This enumeration defines the following values:

CONTINUE	SKIP_SIBLINGS	SKIP_SUBTREE	TERMINATE
----------	---------------	--------------	-----------

In general, to continue traversing the directory and subdirectories, a method should return `CONTINUE`. For `preVisitDirectory()`, return `SKIP_SIBLINGS` to bypass the directory and its siblings and prevent `postVisitDirectory()` from being called. To bypass just the directory and subdirectories, return `SKIP_SUBTREE`. To stop the directory traversal, return `TERMINATE`.

Although it is certainly possible to create your own visitor class that implements these methods defined by `FileVisitor`, you won't normally do so because a simple implementation is provided by `SimpleFileVisitor`. You can just override the default implementation of the method or methods in which you are interested. Here is a short example that illustrates the process. It displays all files in the directory tree that has \MyDir as its root. Notice how short this program is.

```
// A simple example that uses walkFileTree( ) to display a directory tree.
// Requires JDK 7 or later.

import java.io.*;
import java.nio.file.*;
import java.nio.file.attribute.*;

// Create a custom version of SimpleFileVisitor that overrides
// the visitFile( ) method.
class MyFileVisitor extends SimpleFileVisitor<Path> {
    public FileVisitResult visitFile(Path path, BasicFileAttributes attrs)
```

```
        throws IOException
    {
        System.out.println(path);
        return FileVisitResult.CONTINUE;
    }
}

class DirTreeList {
    public static void main(String args[]) {
        String dirname = "\\\MyDir";

        System.out.println("Directory tree starting with " + dirname + ":\n");

        try {
            Files.walkFileTree(Paths.get(dirname), new MyFileVisitor());
        } catch (IOException exc) {
            System.out.println("I/O Error");
        }
    }
}
```

Here is sample output produced by the program when used on the same **MyDir** directory shown earlier. In this example, the subdirectory called **examples** contains one file called **MyProgram.java**.

```
Directory tree starting with \MyDir:
\MyDir\DirList.class
\MyDir\DirList.java
\MyDir\examples\MyProgram.java
\MyDir\Test.txt
```

In the program, the class **MyFileVisitor** extends **SimpleFileVisitor**, overriding only the **visitFile()** method. In this example, **visitFile()** simply displays the files, but more sophisticated functionality is easy to achieve. For example, you could filter the files or perform actions on the files, such as copying them to a backup device. For the sake of clarity, a named class was used to override **visitFile()**, but you could also use an anonymous inner class.

One last point: It is possible to watch a directory for changes by using **java.nio.file.WatchService**.

Pre-JDK 7 Channel-Based Examples

Before concluding this chapter, one more aspect of NIO needs to be covered. The preceding sections have used several of the new features added to NIO by JDK 7. However, you may encounter pre-JDK 7 code that will need to be maintained or possibly converted to use the new features. For this reason, the following sections show how to read and write files using the pre-JDK 7 NIO system. They do so by reworking some of the examples shown earlier so that they use the original NIO features, rather than those supported by NIO.2. This means that the examples in this section will work with versions of Java prior to JDK 7.

The key difference between pre-JDK 7 NIO code and newer NIO code is the **Path** interface, which was added by JDK 7. Thus, pre-JDK 7 code does not use **Path** to describe a file or open a channel to it. Also, pre-JDK 7 code does not use **try-with-resource** statements since automatic resource management was also added by JDK 7.

REMEMBER The examples in this section describe how legacy NIO code works. This section is strictly for the benefit of those programmers working on pre-JDK 7 code or using pre-JDK 7 compilers. New code should take advantage of the NIO features added by JDK 7.

Read a File, Pre-JDK 7

This section reworks the two channel-based file input examples shown earlier so they use only pre-JDK 7 features. The first example reads a file by manually allocating a buffer and then performing an explicit read operation. The second example uses a mapped file, which automates the process.

When using a pre-JDK 7 version of Java to read a file using a channel and a manually allocated buffer, you first open the file for input using **FileInputStream**, using the same mechanism explained in Chapter 20. Next, obtain a channel to this file by calling **getChannel()** on the **FileInputStream** object. It has this general form:

```
FileChannel getChannel()
```

It returns a **FileChannel** object, which encapsulates the channel for file operations. Then, call **allocate()** to allocate a buffer. Because file channels operate on byte buffers, you will use the **allocate()** method defined by **ByteBuffer**, which works as previously described.

The following program shows how to read and display a file called **test.txt** through a channel using explicit input operations for versions of Java prior to JDK 7:

```
// Use Channels to read a file. Pre-JDK 7 version.
import java.io.*;
import java.nio.*;
import java.nio.channels.*;

public class ExplicitChannelRead {
    public static void main(String args[]) {
        FileInputStream fIn = null;
        FileChannel fChan = null;
        ByteBuffer mBuf;
        int count;

        try {
            // First, open a file for input.
            fIn = new FileInputStream("test.txt");

            // Next, obtain a channel to that file.
            fChan = fIn.getChannel();

            // Allocate a buffer.
            mBuf = ByteBuffer.allocate(128);

            do {
```

```
// Read a buffer.  
count = fChan.read(mBuf);  
  
// Stop when end of file is reached.  
if(count != -1) {  
  
    // Rewind the buffer so that it can be read.  
    mBuf.rewind();  
  
    // Read bytes from the buffer and show  
    // them on the screen.  
    for(int i=0; i < count; i++)  
        System.out.print((char)mBuf.get());  
    }  
} while(count != -1);  
  
System.out.println();  
  
} catch (IOException e) {  
    System.out.println("I/O Error " + e);  
} finally {  
    try {  
        if(fChan != null) fChan.close(); // close channel  
    } catch(IOException e) {  
        System.out.println("Error Closing Channel.");  
    }  
    try {  
        if(fIn != null) fIn.close(); // close file  
    } catch(IOException e) {  
        System.out.println("Error Closing File.");  
    }  
}  
}  
}
```

In this program, notice that the file is opened by using the **FileInputStream** constructor, and a reference to that object is assigned to **fIn**. Next, a channel connected to the file is obtained by calling **getChannel()** on **fIn**. After this point, the program works like the NIO.2 version shown previously. To synopsize: The program then calls the **allocate()** method of **ByteBuffer** to allocate a buffer that will hold the contents of the file when it is read. A byte buffer is used because **FileChannel** operates on bytes. A reference to this buffer is stored in **mBuf**. The contents of the file are then read, one buffer at a time, into **mBuf** through a call to **read()**. The number of bytes read is stored in **count**. Next, the buffer is rewound through a call to **rewind()**. This call is necessary because the current position is at the end of the buffer after the call to **read()**, and it must be reset to the start of the buffer in order for the bytes in **mBuf** to be read by calling **get()**. When the end of the file has been reached, the value returned by **read()** will be **-1**. When this occurs, the program ends, explicitly closing the channel and the file.

Another way to read a file is to map it to a buffer. As explained earlier, a principal advantage to this approach is that the buffer automatically contains the contents of the file. No explicit read operation is necessary. To map and read the contents of a file using

pre-JDK 7 NIO, first open the file using **FileInputStream**. Next, obtain a channel to that file by calling **getChannel()** on the file object. Then, map the channel to a buffer by calling **map()** on the **FileChannel** object. The **map()** method works as described earlier.

The following program reworks the preceding example so that it uses only pre-JDK 7 features to create a mapped file:

```
// Use a mapped file to read a file. Pre-JDK 7 version.

import java.io.*;
import java.nio.*;
import java.nio.channels.*;

public class MappedChannelRead {
    public static void main(String args[]) {
        FileInputStream fIn = null;
        FileChannel fChan = null;
        long fSize;
        MappedByteBuffer mBuf;

        try {
            // First, open a file for input.
            fIn = new FileInputStream("test.txt");

            // Next, obtain a channel to that file.
            fChan = fIn.getChannel();

            // Get the size of the file.
            fSize = fChan.size();

            // Now, map the file into a buffer.
            mBuf = fChan.map(FileChannel.MapMode.READ_ONLY, 0, fSize);

            // Read and display bytes from buffer.
            for(int i=0; i < fSize; i++)
                System.out.print((char)mBuf.get());

        } catch (IOException e) {
            System.out.println("I/O Error " + e);
        } finally {
            try {
                if(fChan != null) fChan.close(); // close channel
            } catch(IOException e) {
                System.out.println("Error Closing Channel.");
            }
            try {
                if(fIn != null) fIn.close(); // close file
            } catch(IOException e) {
                System.out.println("Error Closing File.");
            }
        }
    }
}
```

In the program, the file is opened by using the **FileInputStream** constructor, and a reference to that object is assigned to **fIn**. A channel connected to the file is obtained by calling **getChannel()** on **fIn**. Next, the size of the file is obtained. Then, the entire file is mapped into memory by calling **map()**, and a reference to the buffer is stored in **mBuf**. The bytes in **mBuf** are read by calling **get()**.

Write to a File, Pre-JDK 7

This section reworks the two channel-based file output examples shown earlier so that they use only pre-JDK 7 features. The first example writes to a file by manually allocating a buffer and then performing an explicit output operation. The second example uses a mapped file, which automates the process. In both cases, neither **Path** nor **try-with-resources** is used. This is because neither were part of Java until JDK 7.

When using a pre-JDK 7 version of Java to write a file using a channel and a manually allocated buffer, first open the file for output. This is done by creating a **FileOutputStream**, as described in Chapter 20. Next, obtain a channel to the file by calling **getChannel()** and then allocate a byte buffer by calling **allocate()**, as described in the previous section. Next, put the data you want to write into that buffer, and then call **write()** on the channel. The following program demonstrates this procedure. It writes the alphabet to a file called **test.txt**.

```
// Write to a file using NIO. Pre-JDK 7 Version.
import java.io.*;
import java.nio.*;
import java.nio.channels.*;

public class ExplicitChannelWrite {
    public static void main(String args[]) {
        FileOutputStream fOut = null;
        FileChannel fChan = null;
        ByteBuffer mBuf;

        try {
            // First, open the output file.
            fOut = new FileOutputStream("test.txt");

            // Next, get a channel to the output file.
            fChan = fOut.getChannel();

            // Create a buffer.
            mBuf = ByteBuffer.allocate(26);

            // Write some bytes to the buffer.
            for(int i=0; i<26; i++)
                mBuf.put((byte) ('A' + i));

            // Rewind the buffer so that it can be written.
            mBuf.rewind();

            // Write the buffer to the output file.
            fChan.write(mBuf);
        }
    }
}
```

```
    } catch (IOException e) {
        System.out.println("I/O Error " + e);
    } finally {
    try {
        if(fChan != null) fChan.close(); // close channel
    } catch(IOException e) {
        System.out.println("Error Closing Channel.");
    }
    try {
        if(fOut != null) fOut.close(); // close file
    } catch(IOException e) {
        System.out.println("Error Closing File.");
    }
}
}
}
```

The call to **rewind()** on **mBuf** is necessary in order to reset the current position to zero after data has been written to **mBuf**. Remember, each call to **put()** advances the current position. Therefore, it is necessary for the current position to be reset to the start of the buffer before calling **write()**. If this is not done, **write()** will think that there is no data in the buffer.

When using a pre-JDK 7 version of Java to write to a file using a mapped file, follow these steps. First, open the file for read/write operations by creating a **RandomAccessFile** object. This is necessary to enable the file to be both read from and written to. Next, map that file to a buffer by calling **map()** on that object. Then, write to the buffer. Because the buffer is mapped to the file, any changes to that buffer are automatically reflected in the file. Thus, no explicit write operations to the channel are necessary.

Here is the preceding program reworked so that a mapped file is used:

```
// Write to a mapped file. Pre JDK 7 version.
import java.io.*;
import java.nio.*;
import java.nio.channels.*;

public class MappedChannelWrite {
    public static void main(String args[]) {
        RandomAccessFile fOut = null;
        FileChannel fChan = null;
        ByteBuffer mBuf;

        try {
            fOut = new RandomAccessFile("test.txt", "rw");

            // Next, obtain a channel to that file.
            fChan = fOut.getChannel();

            // Then, map the file into a buffer.
            mBuf = fChan.map(FileChannel.MapMode.READ_WRITE, 0, 26);
        }
    }
}
```

```
// Write some bytes to the buffer.  
for(int i=0; i<26; i++)  
    mBuf.put((byte)('A' + i));  
  
} catch (IOException e) {  
    System.out.println("I/O Error " + e);  
} finally {  
    try {  
        if(fChan != null) fChan.close(); // close channel  
    } catch(IOException e) {  
        System.out.println("Error Closing Channel.");  
    }  
    try {  
        if(fOut != null) fOut.close(); // close file  
    } catch(IOException e) {  
        System.out.println("Error Closing File.");  
    }  
}  
}
```

As you can see, there are no explicit write operations to the channel itself. Because **mBuf** is mapped to the file, changes to **mBuf** are automatically reflected in the underlying file.

This page has been intentionally left blank

CHAPTER

22

Networking

As all readers know, Java is practically a synonym for Internet programming. There are a number of reasons for this, not the least of which is its ability to generate secure, cross-platform, portable code. However, one of the most important reasons that Java is the premier language for network programming are the classes defined in the **java.net** package. They provide an easy-to-use means by which programmers of all skill levels can access network resources.

This chapter explores the **java.net** package. It is important to emphasize that networking is a very large and at times complicated topic. It is not possible for this book to discuss all of the capabilities contained in **java.net**. Instead, this chapter focuses on several of its core classes and interfaces.

Networking Basics

Before we begin, it will be useful to review some key networking concepts and terms. At the core of Java's networking support is the concept of a *socket*. A socket identifies an endpoint in a network. The socket paradigm was part of the 4.2BSD Berkeley UNIX release in the early 1980s. Because of this, the term *Berkeley socket* is also used. Sockets are at the foundation of modern networking because a socket allows a single computer to serve many different clients at once, as well as to serve many different types of information. This is accomplished through the use of a *port*, which is a numbered socket on a particular machine. A server process is said to "listen" to a port until a client connects to it. A server is allowed to accept multiple clients connected to the same port number, although each session is unique. To manage multiple client connections, a server process must be multithreaded or have some other means of multiplexing the simultaneous I/O.

Socket communication takes place via a protocol. *Internet Protocol (IP)* is a low-level routing protocol that breaks data into small packets and sends them to an address across a network, which does not guarantee to deliver said packets to the destination. *Transmission Control Protocol (TCP)* is a higher-level protocol that manages to robustly string together these packets, sorting and retransmitting them as necessary to reliably transmit data. A third protocol, *User Datagram Protocol (UDP)*, sits next to TCP and can be used directly to support fast, connectionless, unreliable transport of packets.

Once a connection has been established, a higher-level protocol ensues, which is dependent on which port you are using. TCP/IP reserves the lower 1,024 ports for specific protocols. Many of these will seem familiar to you if you have spent any time surfing the Internet. Port number 21 is for FTP; 23 is for Telnet; 25 is for e-mail; 43 is for whois; 80 is for HTTP; 119 is for netnews—and the list goes on. It is up to each protocol to determine how a client should interact with the port.

For example, HTTP is the protocol that web browsers and servers use to transfer hypertext pages and images. It is a quite simple protocol for a basic page-browsing web server. Here's how it works. When a client requests a file from an HTTP server, an action known as a *hit*, it simply sends the name of the file in a special format to a predefined port and reads back the contents of the file. The server also responds with a status code to tell the client whether or not the request can be fulfilled and why.

A key component of the Internet is the *address*. Every computer on the Internet has one. An Internet address is a number that uniquely identifies each computer on the Net. Originally, all Internet addresses consisted of 32-bit values, organized as four 8-bit values. This address type was specified by IPv4 (Internet Protocol, version 4). However, a new addressing scheme, called IPv6 (Internet Protocol, version 6) has come into play. IPv6 uses a 128-bit value to represent an address, organized into eight 16-bit chunks. Although there are several reasons for and advantages to IPv6, the main one is that it supports a much larger address space than does IPv4. Fortunately, when using Java, you won't normally need to worry about whether IPv4 or IPv6 addresses are used because Java handles the details for you.

Just as the numbers of an IP address describe a network hierarchy, the name of an Internet address, called its *domain name*, describes a machine's location in a name space. For example, **www.HerbSchildt.com** is in the *COM* top-level domain (reserved for U.S. commercial sites); it is called *HerbSchildt*, and *www* identifies the server for web requests. An Internet domain name is mapped to an IP address by the *Domain Naming Service (DNS)*. This enables users to work with domain names, but the Internet operates on IP addresses.

The Networking Classes and Interfaces

Java supports TCP/IP both by extending the already established stream I/O interface introduced in Chapter 20 and by adding the features required to build I/O objects across the network. Java supports both the TCP and UDP protocol families. TCP is used for reliable stream-based I/O across the network. UDP supports a simpler, hence faster, point-to-point datagram-oriented model. The classes contained in the **java.net** package are shown here:

Authenticator	InetAddress	SocketAddress
CacheRequest	InetSocketAddress	SocketImpl
CacheResponse	InterfaceAddress	SocketPermission
ContentHandler	JarURLConnection	StandardSocketOption
CookieHandler	MulticastSocket	URI
CookieManager	NetPermission	URL
DatagramPacket	NetworkInterface	URLClassLoader

DatagramSocket	PasswordAuthentication	URLConnection
DatagramSocketImpl	Proxy	URLDecoder
HttpCookie	ProxySelector	URLEncoder
HttpURLConnection	ResponseCache	URLPermission (Added by JDK 8.)
IDN	SecureCacheResponse	URLStreamHandler
Inet4Address	ServerSocket	
Inet6Address	Socket	

The **java.net** package's interfaces are listed here:

ContentHandlerFactory	FileNameMap	SocketOptions
CookiePolicy	ProtocolFamily	URLStreamHandlerFactory
CookieStore	SocketImplFactory	
DatagramSocketImplFactory	SocketOption	

In the sections that follow, we will examine the main networking classes and show several examples that apply to them. Once you understand these core networking classes, you will be able to easily explore the others on your own.

InetAddress

The **InetAddress** class is used to encapsulate both the numerical IP address and the domain name for that address. You interact with this class by using the name of an IP host, which is more convenient and understandable than its IP address. The **InetAddress** class hides the number inside. **InetAddress** can handle both IPv4 and IPv6 addresses.

Factory Methods

The **InetAddress** class has no visible constructors. To create an **InetAddress** object, you have to use one of the available factory methods. *Factory methods* are merely a convention whereby static methods in a class return an instance of that class. This is done in lieu of overloading a constructor with various parameter lists when having unique method names makes the results much clearer. Three commonly used **InetAddress** factory methods are shown here:

```
static InetAddress getLocalHost( )
    throws UnknownHostException

static InetAddress getByName(String hostName)
    throws UnknownHostException

static InetAddress[ ] getAllByName(String hostName)
    throws UnknownHostException
```

The **getLocalHost()** method simply returns the **InetAddress** object that represents the local host. The **getByName()** method returns an **InetAddress** for a host name passed to it. If these methods are unable to resolve the host name, they throw an **UnknownHostException**.

On the Internet, it is common for a single name to be used to represent several machines. In the world of web servers, this is one way to provide some degree of scaling. The `getAllByName()` factory method returns an array of `InetAddresses` that represent all of the addresses that a particular name resolves to. It will also throw an `UnknownHostException` if it can't resolve the name to at least one address.

`InetAddress` also includes the factory method `getByAddress()`, which takes an IP address and returns an `InetAddress` object. Either an IPv4 or an IPv6 address can be used.

The following example prints the addresses and names of the local machine and two Internet web sites:

```
// Demonstrate InetAddress.
import java.net.*;

class InetAddressTest
{
    public static void main(String args[]) throws UnknownHostException {
        InetAddress Address = InetAddress.getLocalHost();
        System.out.println(Address);

        Address = InetAddress.getByName("www.HerbSchildt.com");
        System.out.println(Address);

        InetAddress SW[] = InetAddress.getAllByName("www.nba.com");
        for (int i=0; i<SW.length; i++)
            System.out.println(SW[i]);
    }
}
```

Here is the output produced by this program. (Of course, the output you see may be slightly different.)

```
default/166.203.115.212
www.HerbSchildt.com/216.92.65.4
www.nba.com/216.66.31.161
www.nba.com/216.66.31.179
```

Instance Methods

The `InetAddress` class has several other methods, which can be used on the objects returned by the methods just discussed. Here are some of the more commonly used methods:

<code>boolean equals(Object other)</code>	Returns <code>true</code> if this object has the same Internet address as <code>other</code> .
<code>byte[] getAddress()</code>	Returns a byte array that represents the object's IP address in network byte order.
<code>String getHostAddress()</code>	Returns a string that represents the host address associated with the <code>InetAddress</code> object.

String getHostName()	Returns a string that represents the host name associated with the InetAddress object.
boolean isMulticastAddress()	Returns true if this address is a multicast address. Otherwise, it returns false .
String toString()	Returns a string that lists the host name and the IP address for convenience.

Internet addresses are looked up in a series of hierarchically cached servers. That means that your local computer might know a particular name-to-IP-address mapping automatically, such as for itself and nearby servers. For other names, it may ask a local DNS server for IP address information. If that server doesn't have a particular address, it can go to a remote site and ask for it. This can continue all the way up to the root server. This process might take a long time, so it is wise to structure your code so that you cache IP address information locally rather than look it up repeatedly.

Inet4Address and Inet6Address

Java includes support for both IPv4 and IPv6 addresses. Because of this, two subclasses of **InetAddress** were created: **Inet4Address** and **Inet6Address**. **Inet4Address** represents a traditional-style IPv4 address. **Inet6Address** encapsulates a newer IPv6 address. Because they are subclasses of **InetAddress**, an **InetAddress** reference can refer to either. This is one way that Java was able to add IPv6 functionality without breaking existing code or adding many more classes. For the most part, you can simply use **InetAddress** when working with IP addresses because it can accommodate both styles.

TCP/IP Client Sockets

TCP/IP sockets are used to implement reliable, bidirectional, persistent, point-to-point, stream-based connections between hosts on the Internet. A socket can be used to connect Java's I/O system to other programs that may reside either on the local machine or on any other machine on the Internet.

NOTE As a general rule, applets may only establish socket connections back to the host from which the applet was downloaded. This restriction exists because it would be dangerous for applets loaded through a firewall to have access to any arbitrary machine.

There are two kinds of TCP sockets in Java. One is for servers, and the other is for clients. The **ServerSocket** class is designed to be a "listener," which waits for clients to connect before doing anything. Thus, **ServerSocket** is for servers. The **Socket** class is for clients. It is designed to connect to server sockets and initiate protocol exchanges. Because client sockets are the most commonly used by Java applications, they are examined here.

The creation of a **Socket** object implicitly establishes a connection between the client and server. There are no methods or constructors that explicitly expose the details of establishing that connection. Here are two constructors used to create client sockets:

Socket(String <i>hostName</i> , int <i>port</i>) throws UnknownHostException, IOException	Creates a socket connected to the named host and port.
Socket(InetAddress <i>ipAddress</i> , int <i>port</i>) throws IOException	Creates a socket using a preexisting InetAddress object and a port.

Socket defines several instance methods. For example, a **Socket** can be examined at any time for the address and port information associated with it, by use of the following methods:

InetAddress getInetAddress()	Returns the InetAddress associated with the Socket object. It returns null if the socket is not connected.
int getPort()	Returns the remote port to which the invoking Socket object is connected. It returns 0 if the socket is not connected.
int getLocalPort()	Returns the local port to which the invoking Socket object is bound. It returns -1 if the socket is not bound.

You can gain access to the input and output streams associated with a **Socket** by use of the **getInputStream()** and **getOutputStream()** methods, as shown here. Each can throw an **IOException** if the socket has been invalidated by a loss of connection. These streams are used exactly like the I/O streams described in Chapter 20 to send and receive data.

InputStream getInputStream() throws IOException	Returns the InputStream associated with the invoking socket.
OutputStream getOutputStream() throws IOException	Returns the OutputStream associated with the invoking socket.

Several other methods are available, including **connect()**, which allows you to specify a new connection; **isConnected()**, which returns true if the socket is connected to a server; **isBound()**, which returns true if the socket is bound to an address; and **isClosed()**, which returns true if the socket is closed. To close a socket, call **close()**. Closing a socket also closes the I/O streams associated with the socket. Beginning with JDK 7, **Socket** also implements **AutoCloseable**, which means that you can use a **try-with-resources** block to manage a socket.

The following program provides a simple **Socket** example. It opens a connection to a "whois" port (port 43) on the InterNIC server, sends the command-line argument down the socket, and then prints the data that is returned. InterNIC will try to look up the argument as a registered Internet domain name, and then send back the IP address and contact information for that site.

```
// Demonstrate Sockets.  
import java.net.*;  
import java.io.*;  
  
class Whois {  
    public static void main(String args[]) throws Exception {  
        int c;  
  
        // Create a socket connected to internic.net, port 43.  
        Socket s = new Socket("whois.internic.net", 43);  
  
        // Obtain input and output streams.  
        InputStream in = s.getInputStream();  
        OutputStream out = s.getOutputStream();  
  
        // Construct a request string.  
  
        String str = (args.length == 0 ? "MHProfessional.com" : args[0]) + "\n";  
        // Convert to bytes.  
        byte buf[] = str.getBytes();  
  
        // Send request.  
        out.write(buf);  
  
        // Read and display response.  
        while ((c = in.read()) != -1) {  
            System.out.print((char) c);  
        }  
        s.close();  
    }  
}
```

If, for example, you obtained information about **MHProfessional.com**, you'd get something similar to the following:

Whois Server Version 2.0

Domain names in the .com and .net domains can now be registered
with many different competing registrars. Go to <http://www.internic.net>
for detailed information.

Domain Name: MHPROFESSIONAL.COM
Registrar: CSC CORPORATE DOMAINS, INC.
Whois Server: whois.corporatedomains.com
Referral URL: <http://www.cscglobal.com>
Name Server: NS1.MHEDU.COM
Name Server: NS2.MHEDU.COM
.
.
.

Here is how the program works. First, a **Socket** is constructed that specifies the host name "whois.internic.net" and the port number 43. **Internic.net** is the InterNIC web site that handles whois requests. Port 43 is the whois port. Next, both input and output streams are opened on the socket. Then, a string is constructed that contains the name of the web site you want to obtain information about. In this case, if no web site is specified on the command line, then "MHProfessional.com" is used. The string is converted into a **byte** array and then sent out of the socket. The response is read by inputting from the socket, and the results are displayed. Finally, the socket is closed, which also closes the I/O streams.

In the preceding example, the socket was closed manually by calling **close()**. If you are using JDK 7 or later, then you can use a **try-with-resources** block to automatically close the socket. For example, here is another way to write the **main()** method of the previous program:

```
// Use try-with-resources to close a socket.
public static void main(String args[]) throws Exception {
    int c;

    // Create a socket connected to internic.net, port 43. Manage this
    // socket with a try-with-resources block.
    try ( Socket s = new Socket("whois.internic.net", 43) ) {

        // Obtain input and output streams.
        InputStream in = s.getInputStream();
        OutputStream out = s.getOutputStream();

        // Construct a request string.
        String str = (args.length == 0 ? "MHProfessional.com" : args[0]) + "\n";
        // Convert to bytes.
        byte buf[] = str.getBytes();

        // Send request.
        out.write(buf);

        // Read and display response.
        while ((c = in.read()) != -1) {
            System.out.print((char) c);
        }
    }
    // The socket is now closed.
}
```

In this version, the socket is automatically closed when the **try** block ends.

So the examples will work with earlier versions of Java and to clearly illustrate when a network resource can be closed, subsequent examples will continue to call **close()** explicitly. However, in your own code, you should consider using automatic resource management since it offers a more streamlined approach. One other point: In this version, exceptions are still thrown out of **main()**, but they could be handled by adding **catch** clauses to the end of the **try-with-resources** block.

NOTE For simplicity, the examples in this chapter simply throw all exceptions out of `main()`. This allows the logic of the network code to be clearly illustrated. However, in real-world code, you will normally need to handle the exceptions in an appropriate way.

URL

The preceding example was rather obscure because the modern Internet is not about the older protocols such as whois, finger, and FTP. It is about WWW, the World Wide Web. The Web is a loose collection of higher-level protocols and file formats, all unified in a web browser. One of the most important aspects of the Web is that Tim Berners-Lee devised a scalable way to locate all of the resources of the Net. Once you can reliably name anything and everything, it becomes a very powerful paradigm. The Uniform Resource Locator (URL) does exactly that.

The URL provides a reasonably intelligible form to uniquely identify or address information on the Internet. URLs are ubiquitous; every browser uses them to identify information on the Web. Within Java's network class library, the `URL` class provides a simple, concise API to access information across the Internet using URLs.

All URLs share the same basic format, although some variation is allowed. Here are two examples: `http://www.MHProfessional.com/` and `http://www.MHProfessional.com:80/index.htm`. A URL specification is based on four components. The first is the protocol to use, separated from the rest of the locator by a colon (:). Common protocols are HTTP, FTP, gopher, and file, although these days almost everything is being done via HTTP (in fact, most browsers will proceed correctly if you leave off the "http://" from your URL specification). The second component is the host name or IP address of the host to use; this is delimited on the left by double slashes (//) and on the right by a slash (/) or optionally a colon (:). The third component, the port number, is an optional parameter, delimited on the left from the host name by a colon (:) and on the right by a slash (/). (It defaults to port 80, the predefined HTTP port; thus, ":80" is redundant.) The fourth part is the actual file path. Most HTTP servers will append a file named `index.html` or `index.htm` to URLs that refer directly to a directory resource. Thus, `http://www.MHProfessional.com/` is the same as `http://www.MHProfessional.com/index.htm`.

Java's `URL` class has several constructors; each can throw a `MalformedURLException`. One commonly used form specifies the URL with a string that is identical to what you see displayed in a browser:

`URL(String urlSpecifier)` throws `MalformedURLException`

The next two forms of the constructor allow you to break up the URL into its component parts:

`URL(String protocolName, String hostName, int port, String path)`
throws `MalformedURLException`

`URL(String protocolName, String hostName, String path)`
throws `MalformedURLException`

Another frequently used constructor allows you to use an existing URL as a reference context and then create a new URL from that context. Although this sounds a little contorted, it's really quite easy and useful.

`URL(URL urlObj, String urlSpecifier) throws MalformedURLException`

The following example creates a URL to a page on **HerbSchildt.com** and then examines its properties:

```
// Demonstrate URL.
import java.net.*;
class URLDemo {
    public static void main(String args[]) throws MalformedURLException {
        URL hp = new URL("http://www.HerbSchildt.com/WhatsNew");

        System.out.println("Protocol: " + hp.getProtocol());
        System.out.println("Port: " + hp.getPort());

        System.out.println("Host: " + hp.getHost());
        System.out.println("File: " + hp.getFile());
        System.out.println("Ext:" + hp.toExternalForm());
    }
}
```

When you run this, you will get the following output:

```
Protocol: http
Port: -1
Host: www.HerbSchildt.com
File: /WhatsNew
Ext: http://www.HerbSchildt.com/WhatsNew
```

Notice that the port is `-1`; this means that a port was not explicitly set. Given a **URL** object, you can retrieve the data associated with it. To access the actual bits or content information of a **URL**, create a **URLConnection** object from it, using its **openConnection()** method, like this:

```
urlc = url.openConnection()
```

openConnection() has the following general form:

`URLConnection openConnection() throws IOException`

It returns a **URLConnection** object associated with the invoking **URL** object. Notice that it may throw an **IOException**.

URLConnection

URLConnection is a general-purpose class for accessing the attributes of a remote resource. Once you make a connection to a remote server, you can use **URLConnection** to inspect the properties of the remote object before actually transporting it locally. These attributes

are exposed by the HTTP protocol specification and, as such, only make sense for **URL** objects that are using the HTTP protocol.

URLConnection defines several methods. Here is a sampling:

<code>int getContentLength()</code>	Returns the size in bytes of the content associated with the resource. If the length is unavailable, -1 is returned.
<code>long getContentLengthLong()</code>	Returns the size in bytes of the content associated with the resource. If the length is unavailable, -1 is returned.
<code>String getContentType()</code>	Returns the type of content found in the resource. This is the value of the content-type header field. Returns null if the content type is not available.
<code>long getDate()</code>	Returns the time and date of the response represented in terms of milliseconds since January 1, 1970 GMT.
<code>long getExpiration()</code>	Returns the expiration time and date of the resource represented in terms of milliseconds since January 1, 1970 GMT. Zero is returned if the expiration date is unavailable.
<code>String getHeaderField(int idx)</code>	Returns the value of the header field at index <i>idx</i> . (Header field indexes begin at 0.) Returns null if the value of <i>idx</i> exceeds the number of fields.
<code>String getHeaderField(String fieldName)</code>	Returns the value of header field whose name is specified by <i>fieldName</i> . Returns null if the specified name is not found.
<code>String getHeaderFieldKey(int idx)</code>	Returns the header field key at index <i>idx</i> . (Header field indexes begin at 0.) Returns null if the value of <i>idx</i> exceeds the number of fields.
<code>Map<String, List<String>> getHeaderFields()</code>	Returns a map that contains all of the header fields and values.
<code>long getLastModified()</code>	Returns the time and date, represented in terms of milliseconds since January 1, 1970 GMT, of the last modification of the resource. Zero is returned if the last-modified date is unavailable.
<code>InputStream getInputStream() throws IOException</code>	Returns an InputStream that is linked to the resource. This stream can be used to obtain the content of the resource.

Notice that **URLConnection** defines several methods that handle header information. A header consists of pairs of keys and values represented as strings. By using `getHeaderField()`, you can obtain the value associated with a header key. By calling `getHeaderFields()`, you can obtain a map that contains all of the headers. Several standard header fields are available directly through methods such as `getDate()` and `getContentType()`.

The following example creates a **URLConnection** using the **openConnection()** method of a **URL** object and then uses it to examine the document's properties and content:

```
// Demonstrate URLConnection.
import java.net.*;
import java.io.*;
import java.util.Date;

class UCDemo
{
    public static void main(String args[]) throws Exception {
        int c;
        URL hp = new URL("http://www.internic.net");
        URLConnection hpCon = hp.openConnection();

        // get date
        long d = hpCon.getDate();
        if(d==0)
            System.out.println("No date information.");
        else
            System.out.println("Date: " + new Date(d));

        // get content type
        System.out.println("Content-Type: " + hpCon.getContentType());

        // get expiration date
        d = hpCon.getExpiration();
        if(d==0)
            System.out.println("No expiration information.");
        else
            System.out.println("Expires: " + new Date(d));

        // get last-modified date
        d = hpCon.getLastModified();
        if(d==0)
            System.out.println("No last-modified information.");
        else
            System.out.println("Last-Modified: " + new Date(d));

        // get content length
        long len = hpCon.getContentLengthLong();
        if(len == -1)
            System.out.println("Content length unavailable.");
        else
            System.out.println("Content-Length: " + len);

        if(len != 0) {
            System.out.println("==== Content ====");
            InputStream input = hpCon.getInputStream();
            while (((c = input.read()) != -1)) {
                System.out.print((char) c);
            }
            input.close();
        }
    }
}
```

```
    } else {
        System.out.println("No content available.");
    }
}
```

The program establishes an HTTP connection to **www.internic.net** over port 80. It then displays several header values and retrieves the content. You might find it interesting to try this example, observing the results, and then for comparison purposes try a different web site of your own choosing.

HttpURLConnection

Java provides a subclass of **URLConnection** that provides support for HTTP connections. This class is called **HttpURLConnection**. You obtain an **HttpURLConnection** in the same way just shown, by calling **openConnection()** on a **URL** object, but you must cast the result to **HttpURLConnection**. (Of course, you must make sure that you are actually opening an HTTP connection.) Once you have obtained a reference to an **HttpURLConnection** object, you can use any of the methods inherited from **URLConnection**. You can also use any of the several methods defined by **HttpURLConnection**. Here is a sampling:

static boolean getFollowRedirects()	Returns true if redirects are automatically followed and false otherwise. This feature is on by default.
String getRequestMethod()	Returns a string representing how URL requests are made. The default is GET. Other options, such as POST, are available.
intgetResponseCode() throws IOException	Returns the HTTP response code. -1 is returned if no response code can be obtained. An IOException is thrown if the connection fails.
StringgetResponseMessage() throws IOException	Returns the response message associated with the response code. Returns null if no message is available. An IOException is thrown if the connection fails.
static void setFollowRedirects(boolean <i>how</i>)	If <i>how</i> is true , then redirects are automatically followed. If <i>how</i> is false , redirects are not automatically followed. By default, redirects are automatically followed.
void setRequestMethod(String <i>how</i>) throws ProtocolException	Sets the method by which HTTP requests are made to that specified by <i>how</i> . The default method is GET, but other options, such as POST, are available. If <i>how</i> is invalid, a ProtocolException is thrown.

The following program demonstrates **HttpURLConnection**. It first establishes a connection to **www.google.com**. Then it displays the request method, the response code, and the response message. Finally, it displays the keys and values in the response header.

```
// Demonstrate HttpURLConnection.
import java.net.*;
import java.io.*;
import java.util.*;

class HttpURLDemo
{

    public static void main(String args[]) throws Exception {
        URL hp = new URL("http://www.google.com");

        HttpURLConnection hpCon = (HttpURLConnection) hp.openConnection();

        // Display request method.
        System.out.println("Request method is " +
                           hpCon.getRequestMethod());

        // Display response code.
        System.out.println("Response code is " +
                           hpCon.getResponseCode());

        // Display response message.
        System.out.println("Response Message is " +
                           hpCon.getResponseMessage());

        // Get a list of the header fields and a set
        // of the header keys.
        Map<String, List<String>> hdrMap = hpCon.getHeaderFields();
        Set<String> hdrField = hdrMap.keySet();

        System.out.println("\nHere is the header:");

        // Display all header keys and values.
        for(String k : hdrField) {
            System.out.println("Key: " + k +
                               " Value: " + hdrMap.get(k));
        }
    }
}
```

The output produced by the program is shown here. (Of course, the exact response returned by **www.google.com** will vary over time.)

```
Request method is GET
Response code is 200
Response Message is OK

Here is the header:
Key: Transfer-Encoding  Value: [chunked]
```

```

Key: X-Frame-Options Value: [SAMEORIGIN]
Key: null Value: [HTTP/1.1 200 OK]
Key: Server Value: [gws]
Key: Cache-Control Value: [private, max-age=0]
Key: Set-Cookie Value:
[NID=67=rMTQWvn5eVIYA2d8F5Iu_8L-68wiMACyaXYqeSe1bvR8SzQQ_PaDCy5mNbzuw5XtdcjY
KIwmy3oVJMJ1Y0qZdibB0kQfJmtHpAtO61GVwumQ1ApgSXWjZ67yHxQX3g3-h; expires=Wed,
23-Apr-2014 18:31:09 GMT; path=/; domain=.google.com; HttpOnly,
PREF=ID=463b5df7b9ced9d8:FF=0:TM=1382466669:LM=1382466669:S=3LI-oT-Dzi46U1On
; expires=Thu, 22-Oct-2015 18:31:09 GMT; path=/; domain=.google.com]
Key: Expires Value: [-1]
Key: X-XSS-Protection Value: [1; mode=block]
Key: P3P Value: [CP="This is not a P3P policy! See
http://www.google.com/support/accounts/bin/answer.py?hl=en&answer=151657 for
more info."]
Key: Date Value: [Tue, 22 Oct 2013 18:31:09 GMT]
Key: Content-Type Value: [text/html; charset=ISO-8859-1]

```

Notice how the header keys and values are displayed. First, a map of the header keys and values is obtained by calling `getHeaderFields()` (which is inherited from `URLConnection`). Next, a set of the header keys is retrieved by calling `keySet()` on the map. Then, the key set is cycled through by using a for-each style `for` loop. The value associated with each key is obtained by calling `get()` on the map.

The URI Class

The `URI` class encapsulates a *Uniform Resource Identifier (URI)*. URIs are similar to URLs. In fact, URLs constitute a subset of URIs. A URI represents a standard way to identify a resource. A URL also describes how to access the resource.

Cookies

The `java.net` package includes classes and interfaces that help manage cookies and can be used to create a stateful (as opposed to stateless) HTTP session. The classes are `CookieHandler`, `CookieManager`, and `HttpCookie`. The interfaces are `CookiePolicy` and `CookieStore`. The creation of a stateful HTTP session is beyond the scope of this book.

NOTE For information about using cookies with servlets, see Chapter 38.

TCP/IP Server Sockets

As mentioned earlier, Java has a different socket class that must be used for creating server applications. The `ServerSocket` class is used to create servers that listen for either local or remote client programs to connect to them on published ports. `ServerSockets` are quite different from normal `Sockets`. When you create a `ServerSocket`, it will register itself with the system as having an interest in client connections. The constructors for `ServerSocket` reflect the port number that you want to accept connections on and, optionally, how long you want the queue for said port to be. The queue length tells the system how many client connections it can leave pending before it should simply refuse connections. The default

is 50. The constructors might throw an **IOException** under adverse conditions. Here are three of its constructors:

<code>ServerSocket(int port) throws IOException</code>	Creates server socket on the specified port with a queue length of 50.
<code>ServerSocket(int port, int maxQueue) throws IOException</code>	Creates a server socket on the specified port with a maximum queue length of <i>maxQueue</i> .
<code>ServerSocket(int port, int maxQueue, InetAddress localAddress) throws IOException</code>	Creates a server socket on the specified port with a maximum queue length of <i>maxQueue</i> . On a multihomed host, <i>localAddress</i> specifies the IP address to which this socket binds.

ServerSocket has a method called **accept()**, which is a blocking call that will wait for a client to initiate communications and then return with a normal **Socket** that is then used for communication with the client.

Datagrams

TCP/IP-style networking is appropriate for most networking needs. It provides a serialized, predictable, reliable stream of packet data. This is not without its cost, however. TCP includes many complicated algorithms for dealing with congestion control on crowded networks, as well as pessimistic expectations about packet loss. This leads to a somewhat inefficient way to transport data. Datagrams provide an alternative.

Datagrams are bundles of information passed between machines. They are somewhat like a hard throw from a well-trained but blindfolded catcher to the third baseman. Once the datagram has been released to its intended target, there is no assurance that it will arrive or even that someone will be there to catch it. Likewise, when the datagram is received, there is no assurance that it hasn't been damaged in transit or that whoever sent it is still there to receive a response.

Java implements datagrams on top of the UDP protocol by using two classes: the **DatagramPacket** object is the data container, while the **DatagramSocket** is the mechanism used to send or receive the **DatagramPackets**. Each is examined here.

DatagramSocket

DatagramSocket defines four public constructors. They are shown here:

```
DatagramSocket( ) throws SocketException
DatagramSocket(int port) throws SocketException
DatagramSocket(int port, InetAddress ipAddress) throws SocketException
DatagramSocket(SocketAddress address) throws SocketException
```

The first creates a **DatagramSocket** bound to any unused port on the local computer. The second creates a **DatagramSocket** bound to the port specified by *port*. The third constructs a **DatagramSocket** bound to the specified port and **InetAddress**. The fourth constructs a **DatagramSocket** bound to the specified **SocketAddress**. **SocketAddress** is an abstract

class that is implemented by the concrete class **InetSocketAddress**. **InetSocketAddress** encapsulates an IP address with a port number. All can throw a **SocketException** if an error occurs while creating the socket.

DatagramSocket defines many methods. Two of the most important are **send()** and **receive()**, which are shown here:

```
void send(DatagramPacket packet) throws IOException
```

```
void receive(DatagramPacket packet) throws IOException
```

The **send()** method sends a packet to the port specified by *packet*. The **receive()** method waits for a packet to be received and returns the result.

DatagramSocket also defines the **close()** method, which closes the socket. Beginning with JDK 7, **DatagramSocket** implements **AutoCloseable**, which means that a **DatagramSocket** can be managed by a **try-with-resources** block.

Other methods give you access to various attributes associated with a **DatagramSocket**. Here is a sampling:

InetAddress getInetAddress()	If the socket is connected, then the address is returned. Otherwise, null is returned.
int getLocalPort()	Returns the number of the local port.
int getPort()	Returns the number of the port to which the socket is connected. It returns -1 if the socket is not connected to a port.
boolean isBound()	Returns true if the socket is bound to an address. Returns false otherwise.
boolean isConnected()	Returns true if the socket is connected to a server. Returns false otherwise.
void setSoTimeout(int <i>millis</i>) throws SocketException	Sets the time-out period to the number of milliseconds passed in <i>millis</i> .

DatagramPacket

DatagramPacket defines several constructors. Four are shown here:

```
DatagramPacket(byte data [ ], int size)
```

```
DatagramPacket(byte data [ ], int offset, int size)
```

```
DatagramPacket(byte data [ ], int size, InetAddress ipAddress, int port)
```

```
DatagramPacket(byte data [ ], int offset, int size, InetAddress ipAddress, int port)
```

The first constructor specifies a buffer that will receive data and the size of a packet. It is used for receiving data over a **DatagramSocket**. The second form allows you to specify an offset into the buffer at which data will be stored. The third form specifies a target address and port, which are used by a **DatagramSocket** to determine where the data in the packet will be sent. The fourth form transmits packets beginning at the specified offset into the data. Think of the first two forms as building an "in box," and the second two forms as stuffing and addressing an envelope.

DatagramPacket defines several methods, including those shown here, that give access to the address and port number of a packet, as well as the raw data and its length.

InetAddress getAddress()	Returns the address of the source (for datagrams being received) or destination (for datagrams being sent).
byte[] getData()	Returns the byte array of data contained in the datagram. Mostly used to retrieve data from the datagram after it has been received.
int getLength()	Returns the length of the valid data contained in the byte array that would be returned from the <code>getData()</code> method. This may not equal the length of the whole byte array.
int getOffset()	Returns the starting index of the data.
int getPort()	Returns the port number.
void setAddress(InetAddress <i>ipAddress</i>)	Sets the address to which a packet will be sent. The address is specified by <i>ipAddress</i> .
void setData(byte[] <i>data</i>)	Sets the data to <i>data</i> , the offset to zero, and the length to number of bytes in <i>data</i> .
void setData(byte[] <i>data</i> , int <i>idx</i> , int <i>size</i>)	Sets the data to <i>data</i> , the offset to <i>idx</i> , and the length to <i>size</i> .
void setLength(int <i>size</i>)	Sets the length of the packet to <i>size</i> .
void setPort(int <i>port</i>)	Sets the port to <i>port</i> .

A Datagram Example

The following example implements a very simple networked communications client and server. Messages are typed into the window at the server and written across the network to the client side, where they are displayed.

```
// Demonstrate datagrams.
import java.net.*;

class WriteServer {
    public static int serverPort = 998;
    public static int clientPort = 999;
    public static int buffer_size = 1024;
    public static DatagramSocket ds;
    public static byte buffer[] = new byte[buffer_size];

    public static void TheServer() throws Exception {
        int pos=0;
        while (true) {
            int c = System.in.read();
            switch (c) {
                case -1:
                    System.out.println("Server Quits.");
                    break;
            }
        }
    }
}
```

```

        ds.close();
        return;
    case '\r':
        break;
    case '\n':
        ds.send(new DatagramPacket(buffer, pos,
            InetAddress.getLocalHost(), clientPort));
        pos=0;
        break;
    default:
        buffer[pos++] = (byte) c;
    }
}
}

public static void TheClient() throws Exception {
    while(true) {
        DatagramPacket p = new DatagramPacket(buffer, buffer.length);
        ds.receive(p);
        System.out.println(new String(p.getData(), 0, p.getLength()));
    }
}

public static void main(String args[]) throws Exception {
    if(args.length == 1) {
        ds = new DatagramSocket(serverPort);
        TheServer();
    } else {
        ds = new DatagramSocket(clientPort);
        TheClient();
    }
}
}

```

This sample program is restricted by the **DatagramSocket** constructor to running between two ports on the local machine. To use the program, run

`java WriteServer`

in one window; this will be the client. Then run

`java WriteServer 1`

This will be the server. Anything that is typed in the server window will be sent to the client window after a newline is received.

NOTE The use of datagrams may not be allowed on your computer. (For example, a firewall may prevent their use.) If this is the case, the preceding example cannot be used. Also, the port numbers used in the program work on the author's system, but may have to be adjusted for your environment.

This page has been intentionally left blank

CHAPTER

23

The Applet Class

This chapter examines the **Applet** class, which provides the foundation for applets. The **Applet** class is contained in the **java.applet** package. **Applet** contains several methods that give you detailed control over the execution of your applet. In addition, **java.applet** also defines three interfaces: **AppletContext**, **AudioClip**, and **AppletStub**.

Two Types of Applets

It is important to state at the outset that there are two varieties of applets based on **Applet**. The first are those based directly on the **Applet** class described in this chapter. These applets use the Abstract Window Toolkit (AWT) to provide the graphical user interface (or use no GUI at all). This style of applet has been available since Java was first created.

The second type of applets are those based on the Swing class **JApplet**, which inherits **Applet**. Swing applets use the Swing classes to provide the GUI. Swing offers a richer and often easier-to-use user interface than does the AWT. Thus, Swing-based applets are now the most popular. However, traditional AWT-based applets are still used, especially when only a very simple user interface is required. Thus, both AWT- and Swing-based applets are valid.

This chapter describes AWT-based applets. However, because **JApplet** inherits **Applet**, all the features of **Applet** are also available in **JApplet**, and much of the information in this chapter applies to both types of applets. Therefore, even if you are interested in only Swing applets, the information in this chapter is still relevant and necessary. Understand, however, that when creating Swing-based applets, some additional constraints apply and these are described later in this book, when Swing is covered.

NOTE For information on building applets when using Swing, see Chapter 31.

Applet Basics

Chapter 13 introduced the general form of an applet and the steps necessary to compile and run one. Let's begin by reviewing this information.

AWT-based applets are subclasses of **Applet**. Applets are not stand-alone programs. Instead, they run within either a web browser or an applet viewer. The illustrations shown in this chapter were created with the standard applet viewer, called **appletviewer**, provided by the JDK.

Execution of an applet does not begin at `main()`. Actually, few applets even have `main()` methods. Instead, execution of an applet is started and controlled with an entirely different mechanism, which will be explained shortly. Output to your applet's window is not performed by `System.out.println()`. Rather, in an AWT-based applet, output is handled with various AWT methods, such as `drawString()`, which outputs a string to a specified X,Y location. Input is also handled differently than in a console application.

Before an applet can be used, a deployment strategy must be chosen. There are two basic approaches. The first is to use the Java Network Launch Protocol (JNLP). This approach offers the most flexibility, especially as it relates to rich Internet applications. For real-world applets that you create, JNLP will often be the best choice. However, a detailed discussion of JNLP is beyond the scope of this book. (See the JDK documentation for the latest details on JNLP.) Fortunately, JNLP is not required for the example applets shown here.

The second basic approach to deploying an applet is to specify the applet directly in an HTML file, without the use of JNLP. This is the original way that applets were launched when Java was created, and it is still used today—especially for simple applets. Furthermore, because of its inherent simplicity, it is the appropriate method for the applet examples described in this book. At the time of this writing, Oracle recommends the APPLET tag for this purpose. Therefore, the APPLET tag is used in this book. (Be aware that the APPLET tag is currently deprecated by the HTML specification. The alternative is the OBJECT tag. You should check the JDK documentation in this regard for the latest recommendations.) When an APPLET tag is encountered in the HTML file, the specified applet will be executed by a Java-enabled web browser.

The use of the APPLET tag offers a secondary advantage when developing applets because it enables you to easily view and test the applet. To do so, simply include a comment at the head of your Java source code file that contains the APPLET tag. This way, your code is documented with the necessary HTML statements needed by your applet, and you can test the compiled applet by starting the applet viewer with your Java source code file specified as the target. Here is an example of such a comment:

```
/*
<applet code="MyApplet" width=200 height=60>
</applet>
*/
```

This comment contains an APPLET tag that will run an applet called **MyApplet** in a window that is 200 pixels wide and 60 pixels high. Because the inclusion of an APPLET command makes testing applets easier, all of the applets shown in this book will contain the appropriate APPLET tag embedded in a comment.

NOTE As noted in Chapter 13, beginning with the release of Java 7, update 21, Java applets must be signed to prevent security warnings when run in a browser. In fact, in some cases, the applet may be prevented from running. Applets stored in the local file system, such as you would create when compiling the examples in this book, are especially sensitive to this change. You may need to adjust the security settings in the Java Control Panel to run a local applet in a browser. At the time of this writing, Oracle recommends against the use of local applets, recommending instead that applets be executed through a web server. Furthermore, unsigned local applets may be blocked from execution in the future. In general, for applets that will be distributed via the Internet, such as commercial

applications, signing is a virtual necessity. The concepts and techniques required to sign applets (and other types of Java programs) are beyond the scope of this book. However, extensive information is found on Oracle's website. Finally, as mentioned, the easiest way to try the applet examples is to use **appletviewer**.

The Applet Class

The **Applet** class defines the methods shown in Table 23-1. **Applet** provides all necessary support for applet execution, such as starting and stopping. It also provides methods that load and display images, and methods that load and play audio clips. **Applet** extends the AWT class **Panel**. In turn, **Panel** extends **Container**, which extends **Component**. These classes provide support for Java's window-based, graphical interface. Thus, **Applet** provides all of the necessary support for window-based activities. (An overview of the AWT is presented in subsequent chapters.)

Method	Description
void <code>destroy()</code>	Called by the browser just before an applet is terminated. Your applet will override this method if it needs to perform any cleanup prior to its destruction.
<code>AccessibleContext getAccessibleContext()</code>	Returns the accessibility context for the invoking object.
<code>AppletContext getAppletContext()</code>	Returns the context associated with the applet.
<code>String getAppletInfo()</code>	Overrides of this method should return a string that describes the applet. The default implementation returns <code>null</code> .
<code>AudioClip getAudioClip(URL url)</code>	Returns an AudioClip object that encapsulates the audio clip found at the location specified by <i>url</i> .
<code>AudioClip getAudioClip(URL url, String clipName)</code>	Returns an AudioClip object that encapsulates the audio clip found at the location specified by <i>url</i> and having the name specified by <i>clipName</i> .
<code>URL getCodeBase()</code>	Returns the URL associated with the invoking applet.
<code>URL getDocumentBase()</code>	Returns the URL of the HTML document that invokes the applet.
<code>Image getImage(URL url)</code>	Returns an Image object that encapsulates the image found at the location specified by <i>url</i> .
<code>Image getImage(URL url, String imageName)</code>	Returns an Image object that encapsulates the image found at the location specified by <i>url</i> and having the name specified by <i>imageName</i> .
<code>Locale getLocale()</code>	Returns a Locale object that is used by various locale-sensitive classes and methods.
<code>String getParameter(String paramName)</code>	Returns the parameter associated with <i>paramName</i> . null is returned if the specified parameter is not found.

Table 23-1 The Methods Defined by **Applet**

Method	Description
String[] [] <code>getParameterInfo()</code>	Overrides of this method should return a String table that describes the parameters recognized by the applet. Each entry in the table must consist of three strings that contain the name of the parameter, a description of its type and/or range, and an explanation of its purpose. The default implementation returns null .
<code>void init()</code>	Called when an applet begins execution. It is the first method called for any applet.
<code>boolean isActive()</code>	Returns true if the applet has been started. It returns false if the applet has been stopped.
<code>boolean isValidateRoot()</code>	Returns true , which indicates that an applet is a validate root.
<code>static final AudioClip newAudioClip(URL url)</code>	Returns an AudioClip object that encapsulates the audio clip found at the location specified by <i>url</i> . This method is similar to <code>getAudioClip()</code> except that it is static and can be executed without the need for an Applet object.
<code>void play(URL url)</code>	If an audio clip is found at the location specified by <i>url</i> , the clip is played.
<code>void play(URL url, String clipName)</code>	If an audio clip is found at the location specified by <i>url</i> with the name specified by <i>clipName</i> , the clip is played.
<code>void resize(Dimension dim)</code>	Resizes the applet according to the dimensions specified by <i>dim</i> . Dimension is a class stored inside <code>java.awt</code> . It contains two integer fields: width and height .
<code>void resize(int width, int height)</code>	Resizes the applet according to the dimensions specified by <i>width</i> and <i>height</i> .
<code>final void setStub(AppletStub stubObj)</code>	Makes <i>stubObj</i> the stub for the applet. This method is used by the run-time system and is not usually called by your applet. A <i>stub</i> is a small piece of code that provides the linkage between your applet and the browser.
<code>void showStatus(String str)</code>	Displays <i>str</i> in the status window of the browser or applet viewer. If the browser does not support a status window, then no action takes place.
<code>void start()</code>	Called by the browser when an applet should start (or resume) execution. It is automatically called after <code>init()</code> when an applet first begins.
<code>void stop()</code>	Called by the browser to suspend execution of the applet. Once stopped, an applet is restarted when the browser calls <code>start()</code> .

Table 23-1 The Methods Defined by **Applet** (*continued*)

Applet Architecture

As a general rule, an applet is a GUI-based program. As such, its architecture is different from the console-based programs shown in the first part of this book. If you are already familiar with GUI programming, you will be right at home writing applets. If not, then there are a few key concepts you must understand.

First, applets are event driven. Although we won't examine event handling until the following chapter, it is important to understand in a general way how the event-driven architecture impacts the design of an applet. An applet resembles a set of interrupt service routines. Here is how the process works. An applet waits until an event occurs. The run-time system notifies the applet about an event by calling an event handler that has been provided by the applet. Once this happens, the applet must take appropriate action and then quickly return. This is a crucial point. For the most part, your applet should not enter a "mode" of operation in which it maintains control for an extended period. Instead, it must perform specific actions in response to events and then return control to the run-time system. In those situations in which your applet needs to perform a repetitive task on its own (for example, displaying a scrolling message across its window), you must start an additional thread of execution. (You will see an example later in this chapter.)

Second, the user initiates interaction with an applet—not the other way around. As you know, in a console-based program, when the program needs input, it will prompt the user and then call some input method, such as `readLine()`. This is not the way it works in an applet. Instead, the user interacts with the applet as he or she wants, when he or she wants. These interactions are sent to the applet as events to which the applet must respond. For example, when the user clicks the mouse inside the applet's window, a mouse-clicked event is generated. If the user presses a key while the applet's window has input focus, a keypress event is generated. As you will see in later chapters, applets can contain various controls, such as push buttons and check boxes. When the user interacts with one of these controls, an event is generated.

While the architecture of an applet is not as easy to understand as that of a console-based program, Java makes it as simple as possible. If you have written programs for Windows (or other GUI-based operating systems), you know how intimidating that environment can be. Fortunately, Java provides a much cleaner approach that is more quickly mastered.

An Applet Skeleton

All but the most trivial applets override a set of methods that provides the basic mechanism by which the browser or applet viewer interfaces to the applet and controls its execution. Four of these methods, `init()`, `start()`, `stop()`, and `destroy()`, apply to all applets and are defined by `Applet`. Default implementations for all of these methods are provided. Applets do not need to override those methods they do not use. However, only very simple applets will not need to define all of them.

AWT-based applets (such as those discussed in this chapter) will also often override the `paint()` method, which is defined by the AWT `Component` class. This method is called when

the applet's output must be redisplayed. (Swing-based applets use a different mechanism to accomplish this task.) These five methods can be assembled into the skeleton shown here:

```
// An Applet skeleton.
import java.awt.*;
import java.applet.*;
/*
<applet code="AppletSkel" width=300 height=100>
</applet>
*/

public class AppletSkel extends Applet {
    // Called first.
    public void init() {
        // initialization
    }

    /* Called second, after init().  Also called whenever
       the applet is restarted. */
    public void start() {
        // start or resume execution
    }

    // Called when the applet is stopped.
    public void stop() {
        // suspends execution
    }

    /* Called when applet is terminated.  This is the last
       method executed. */
    public void destroy() {
        // perform shutdown activities
    }

    // Called when an applet's window must be restored.
    public void paint(Graphics g) {
        // redisplay contents of window
    }
}
```

Although this skeleton does not do anything, it can be compiled and run. When run, it generates the following empty window when viewed with **appletviewer**. Of course, in this and all subsequent examples, the precise look of the **appletviewer** frame may differ based on your execution environment. To help illustrate this fact, a variety of environments were used to generate the screen captures shown throughout this book.



Applet Initialization and Termination

It is important to understand the order in which the various methods shown in the skeleton are called. When an applet begins, the following methods are called, in this sequence:

1. **init()**
2. **start()**
3. **paint()**

When an applet is terminated, the following sequence of method calls takes place:

1. **stop()**
2. **destroy()**

Let's look more closely at these methods.

init()

The **init()** method is the first method to be called. This is where you should initialize variables. This method is called only once during the run time of your applet.

start()

The **start()** method is called after **init()**. It is also called to restart an applet after it has been stopped. Whereas **init()** is called once—the first time an applet is loaded—**start()** is called each time an applet's HTML document is displayed onscreen. So, if a user leaves a web page and comes back, the applet resumes execution at **start()**.

paint()

The **paint()** method is called each time an AWT-based applet's output must be redrawn. This situation can occur for several reasons. For example, the window in which the applet is running may be overwritten by another window and then uncovered. Or the applet window may be minimized and then restored. **paint()** is also called when the applet begins execution. Whatever the cause, whenever the applet must redraw its output, **paint()** is called. The **paint()** method has one parameter of type **Graphics**. This parameter will contain the graphics context, which describes the graphics environment in which the applet is running. This context is used whenever output to the applet is required.

stop()

The **stop()** method is called when a web browser leaves the HTML document containing the applet—when it goes to another page, for example. When **stop()** is called, the applet is probably running. You should use **stop()** to suspend threads that don't need to run when the applet is not visible. You can restart them when **start()** is called if the user returns to the page.

destroy()

The **destroy()** method is called when the environment determines that your applet needs to be removed completely from memory. At this point, you should free up any resources the applet may be using. The **stop()** method is always called before **destroy()**.

Overriding update()

In some situations, an AWT-based applet may need to override another method defined by the AWT, called **update()**. This method is called when your applet has requested that a portion of its window be redrawn. The default version of **update()** simply calls **paint()**. However, you can override the **update()** method so that it performs more subtle repainting. In general, overriding **update()** is a specialized technique that is not applicable to all applets, and the examples in this chapter do not override **update()**.

Simple Applet Display Methods

As we've mentioned, applets are displayed in a window, and AWT-based applets use the AWT to perform input and output. Although we will examine the methods, procedures, and techniques related to the AWT in subsequent chapters, a few are described here, because we will use them to write sample applets. (Remember, Swing-based applets are described later in this book.)

As described in Chapter 13, to output a string to an applet, use **drawString()**, which is a member of the **Graphics** class. Typically, it is called from within either **update()** or **paint()**. It has the following general form:

```
void drawString(String message, int x, int y)
```

Here, *message* is the string to be output beginning at *x,y*. In a Java window, the upper-left corner is location 0,0. The **drawString()** method will not recognize newline characters. If you want to start a line of text on another line, you must do so manually, specifying the precise X,Y location where you want the line to begin. (As you will see in later chapters, there are techniques that make this process easy.)

To set the background color of an applet's window, use **setBackground()**. To set the foreground color (the color in which text is shown, for example), use **setForeground()**. These methods are defined by **Component**, and they have the following general forms:

```
void setBackground(Color newColor)
void setForeground(Color newColor)
```

Here, *newColor* specifies the new color. The class **Color** defines the constants shown here that can be used to specify colors:

Color.black	Color.magenta
Color.blue	Color.orange
Color.cyan	Color.pink
Color.darkGray	Color.red
Color.gray	Color.white
Color.green	Color.yellow
Color.lightGray	

Uppercase versions of the constants are also defined.

The following example sets the background color to green and the text color to red:

```
setBackground(Color.green);
setForeground(Color.red);
```

A good place to set the foreground and background colors is in the **init()** method. Of course, you can change these colors as often as necessary during the execution of your applet.

You can obtain the current settings for the background and foreground colors by calling **getBackground()** and **getForeground()**, respectively. They are also defined by **Component** and are shown here:

```
Color getBackground()
Color getForeground()
```

Here is a very simple applet that sets the background color to cyan, the foreground color to red, and displays a message that illustrates the order in which the **init()**, **start()**, and **paint()** methods are called when an applet starts up:

```
/* A simple applet that sets the foreground and
background colors and outputs a string. */
import java.awt.*;
import java.applet.*;
/*
<applet code="Sample" width=300 height=50>
</applet>
*/
public class Sample extends Applet{
    String msg;

    // set the foreground and background colors.
    public void init() {
        setBackground(Color.cyan);
        setForeground(Color.red);
        msg = "Inside init( ) --";
    }

    // Initialize the string to be displayed.
    public void start() {
        msg += " Inside start( ) --";
    }

    // Display msg in applet window.
    public void paint(Graphics g) {
        msg += " Inside paint( ).";
        g.drawString(msg, 10, 30);
    }
}
```

Sample output is shown here:



The methods **stop()** and **destroy()** are not overridden, because they are not needed by this simple applet.

Requesting Repainting

As a general rule, an applet writes to its window only when its **paint()** method is called by the AWT. This raises an interesting question: How can the applet itself cause its window to be updated when its information changes? For example, if an applet is displaying a moving banner, what mechanism does the applet use to update the window each time this banner scrolls? Remember, one of the fundamental architectural constraints imposed on an applet is that it must quickly return control to the run-time system. It cannot create a loop inside **paint()** that repeatedly scrolls the banner, for example. This would prevent control from passing back to the AWT. Given this constraint, it may seem that output to your applet's window will be difficult at best. Fortunately, this is not the case. Whenever your applet needs to update the information displayed in its window, it simply calls **repaint()**.

The **repaint()** method is defined by the AWT. It causes the AWT run-time system to execute a call to your applet's **update()** method, which, in its default implementation, calls **paint()**. Thus, for another part of your applet to output to its window, simply store the output and then call **repaint()**. The AWT will then execute a call to **paint()**, which can display the stored information. For example, if part of your applet needs to output a string, it can store this string in a **String** variable and then call **repaint()**. Inside **paint()**, you will output the string using **drawString()**.

The **repaint()** method has four forms. Let's look at each one, in turn. The simplest version of **repaint()** is shown here:

```
void repaint()
```

This version causes the entire window to be repainted. The following version specifies a region that will be repainted:

```
void repaint(int left, int top, int width, int height)
```

Here, the coordinates of the upper-left corner of the region are specified by *left* and *top*, and the width and height of the region are passed in *width* and *height*. These dimensions are specified in pixels. You save time by specifying a region to repaint. Window updates are costly in terms of time. If you need to update only a small portion of the window, it is more efficient to repaint only that region.

Calling **repaint()** is essentially a request that your applet be repainted sometime soon. However, if your system is slow or busy, **update()** might not be called immediately. Multiple requests for repainting that occur within a short time can be collapsed by the AWT in a manner such that **update()** is only called sporadically. This can be a problem in many situations, including animation, in which a consistent update time is necessary. One solution to this problem is to use the following forms of **repaint()**:

```
void repaint(long maxDelay)
void repaint(long maxDelay, int x, int width, int height)
```

Here, *maxDelay* specifies the maximum number of milliseconds that can elapse before **update()** is called. Beware, though. If the time elapses before **update()** can be called, it isn't called. There's no return value or exception thrown, so you must be careful.

NOTE It is possible for a method other than **paint()** or **update()** to output to an applet's window. To do so, it must obtain a graphics context by calling **getGraphics()** (defined by **Component**) and then use this context to output to the window. However, for most applications, it is better and easier to route window output through **paint()** and to call **repaint()** when the contents of the window change.

A Simple Banner Applet

To demonstrate **repaint()**, a simple banner applet is developed. This applet scrolls a message, from right to left, across the applet's window. Since the scrolling of the message is a repetitive task, it is performed by a separate thread, created by the applet when it is initialized. The banner applet is shown here:

```
/* A simple banner applet.

This applet creates a thread that scrolls
the message contained in msg right to left
across the applet's window.

*/
import java.awt.*;
import java.applet.*;
/*
<applet code="SimpleBanner" width=300 height=50>
</applet>
*/

public class SimpleBanner extends Applet implements Runnable {
    String msg = " A Simple Moving Banner.";
    Thread t = null;
    int state;
    volatile boolean stopFlag;

    // Set colors and initialize thread.
    public void init() {
        setBackground(Color.cyan);
        setForeground(Color.red);
    }
}
```

```
// Start thread
public void start() {
    t = new Thread(this);
    stopFlag = false;
    t.start();
}

// Entry point for the thread that runs the banner.
public void run() {

    // Redisplay banner
    for( ; ; ) {
        try {
            repaint();
            Thread.sleep(250);
            if(stopFlag)
                break;
        } catch(InterruptedException e) {}
    }
}

// Pause the banner.
public void stop() {
    stopFlag = true;
    t = null;
}

// Display the banner.
public void paint(Graphics g) {
    char ch;

    ch = msg.charAt(0);
    msg = msg.substring(1, msg.length());
    msg += ch;

    g.drawString(msg, 50, 30);
}
}
```

Following is sample output:



Let's take a close look at how this applet operates. First, notice that **SimpleBanner** extends **Applet**, as expected, but it also implements **Runnable**. This is necessary, since the

applet will be creating a second thread of execution that will be used to scroll the banner. Inside `init()`, the foreground and background colors of the applet are set.

After initialization, the run-time system calls `start()` to start the applet running. Inside `start()`, a new thread of execution is created and assigned to the **Thread** variable `t`. Then, the **boolean** variable `stopFlag`, which controls the execution of the applet, is set to **false**. Next, the thread is started by a call to `t.start()`. Remember that `t.start()` calls a method defined by **Thread**, which causes `run()` to begin executing. It does not cause a call to the version of `start()` defined by **Applet**. These are two separate methods.

Inside `run()`, a call to `repaint()` is made. This eventually causes the `paint()` method to be called, and the rotated contents of `msg` are displayed. Between each iteration, `run()` sleeps for a quarter of a second. The net effect is that the contents of `msg` are scrolled right to left in a constantly moving display. The `stopFlag` variable is checked on each iteration. When it is **true**, the `run()` method terminates.

If a browser is displaying the applet when a new page is viewed, the `stop()` method is called, which sets `stopFlag` to **true**, causing `run()` to terminate. This is the mechanism used to stop the thread when its page is no longer in view. When the applet is brought back into view, `start()` is once again called, which starts a new thread to execute the banner.

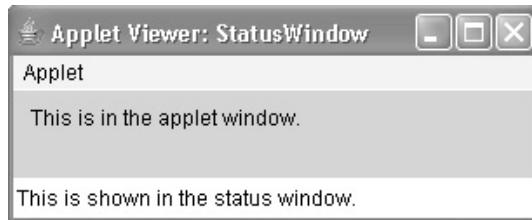
Using the Status Window

In addition to displaying information in its window, an applet can also output a message to the status window of the browser or applet viewer on which it is running. To do so, call `showStatus()` with the string that you want displayed. The status window is a good place to give the user feedback about what is occurring in the applet, suggest options, or possibly report some types of errors. The status window also makes an excellent debugging aid, because it gives you an easy way to output information about your applet.

The following applet demonstrates `showStatus()`:

```
// Using the Status Window.
import java.awt.*;
import java.applet.*;
/*
<applet code="StatusWindow" width=300 height=50>
</applet>
*/
public class StatusWindow extends Applet {
    public void init() {
        setBackground(Color.cyan);
    }
    // Display msg in applet window.
    public void paint(Graphics g) {
        g.drawString("This is in the applet window.", 10, 20);
        showStatus("This is shown in the status window.");
    }
}
```

Sample output from this program is shown here:



The HTML APPLET Tag

As mentioned earlier, at the time of this writing, Oracle recommends that the APPLET tag be used to manually start an applet when JNLP is not used. An applet viewer will execute each APPLET tag that it finds in a separate window, while web browsers will allow many applets on a single page. So far, we have been using only a simplified form of the APPLET tag. Now it is time to take a closer look at it.

The syntax for a fuller form of the APPLET tag is shown here. Bracketed items are optional.

```
<APPLET
  [CODEBASE = codebaseURL]
  CODE = appletFile
  [ALT = alternateText]
  [NAME = appletInstanceName]
  WIDTH = pixels HEIGHT = pixels
  [ALIGN = alignment ]
  [VSPACE = pixels] [HSPACE = pixels]
>
[< PARAM NAME = AttributeName VALUE = AttributeValue>]
[< PARAM NAME = AttributeName2 VALUE = AttributeValue>]
...
[HTML Displayed in the absence of Java]
</APPLET>
```

Let's take a look at each part now.

CODEBASE CODEBASE is an optional attribute that specifies the base URL of the applet code, which is the directory that will be searched for the applet's executable class file (specified by the CODE tag). The HTML document's URL directory is used as the CODEBASE if this attribute is not specified.

CODE CODE is a required attribute that gives the name of the file containing your applet's compiled **.class** file. This file is relative to the code base URL of the applet, which is the directory that the HTML file was in or the directory indicated by CODEBASE if set.

ALT The ALT tag is an optional attribute used to specify a short text message that should be displayed if the browser recognizes the APPLET tag but can't currently run Java applets. This is distinct from the alternate HTML you provide for browsers that don't support applets.

NAME NAME is an optional attribute used to specify a name for the applet instance. Applets must be named in order for other applets on the same page to find them by name and communicate with them. To obtain an applet by name, use `getApplet()`, which is defined by the `AppletContext` interface.

WIDTH and HEIGHT WIDTH and HEIGHT are required attributes that give the size (in pixels) of the applet display area.

ALIGN ALIGN is an optional attribute that specifies the alignment of the applet. This attribute is treated the same as the HTML IMG tag with these possible values: LEFT, RIGHT, TOP, BOTTOM, MIDDLE, BASELINE, TEXTTOP, ABSMIDDLE, and ABSBOTTOM.

VSPACE and HSPACE These attributes are optional. VSPACE specifies the space, in pixels, above and below the applet. HSPACE specifies the space, in pixels, on each side of the applet. They're treated the same as the IMG tag's VSPACE and HSPACE attributes.

PARAM NAME and VALUE The PARAM tag allows you to specify applet-specific arguments. Applets access their attributes with the `getParameter()` method.

Other valid APPLET attributes include ARCHIVE, which lets you specify one or more archive files, and OBJECT, which specifies a saved version of the applet. In general, an APPLET tag should include only a CODE or an OBJECT attribute, but not both.

Passing Parameters to Applets

As just discussed, the APPLET tag allows you to pass parameters to your applet. To retrieve a parameter, use the `getParameter()` method. It returns the value of the specified parameter in the form of a `String` object. Thus, for numeric and `boolean` values, you will need to convert their string representations into their internal formats. Here is an example that demonstrates passing parameters:

```
// Use Parameters
import java.awt.*;
import java.applet.*;
/*
<applet code="ParamDemo" width=300 height=80>
<param name=fontName value=Courier>
<param name=fontSize value=14>
<param name=leading value=2>
<param name=accountEnabled value=true>
</applet>
*/
public class ParamDemo extends Applet {
    String fontName;
```

```

int fontSize;
float leading;
boolean active;

// Initialize the string to be displayed.
public void start() {
    String param;

    fontName = getParameter("fontName");
    if(fontName == null)
        fontName = "Not Found";

    param = getParameter("fontSize");
    try {
        if(param != null)
            fontSize = Integer.parseInt(param);
        else
            fontSize = 0;
    } catch(NumberFormatException e) {
        fontSize = -1;
    }

    param = getParameter("leading");
    try {
        if(param != null)
            leading = Float.valueOf(param).floatValue();
        else
            leading = 0;
    } catch(NumberFormatException e) {
        leading = -1;
    }

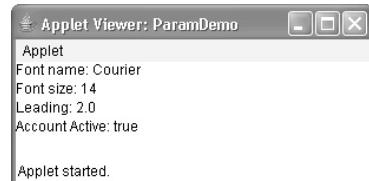
    param = getParameter("accountEnabled");
    if(param != null)
        active = Boolean.valueOf(param).booleanValue();
    }

// Display parameters.
public void paint(Graphics g) {
    g.drawString("Font name: " + fontName, 0, 10);
    g.drawString("Font size: " + fontSize, 0, 26);
    g.drawString("Leading: " + leading, 0, 42);
    g.drawString("Account Active: " + active, 0, 58);
}
}

```

Sample output from this program is shown here:

As the program shows, you should test the return values from `getParameter()`. If a parameter isn't available, `getParameter()` will return `null`. Also, conversions to numeric types must be attempted in a `try` statement that catches `NumberFormatException`. Uncaught exceptions should never occur within an applet.



Improving the Banner Applet

It is possible to use a parameter to enhance the banner applet shown earlier. In the previous version, the message being scrolled was hard-coded into the applet. However, passing the message as a parameter allows the banner applet to display a different message each time it is executed. This improved version is shown here. Notice that the APPLET tag at the top of the file now specifies a parameter called **message** that is linked to a quoted string.

```
// A parameterized banner
import java.awt.*;
import java.applet.*;
/*
<applet code="ParamBanner" width=300 height=50>
<param name=message value="Java makes the Web move!">
</applet>
*/
public class ParamBanner extends Applet implements Runnable {
    String msg;
    Thread t = null;
    int state;
    volatile boolean stopFlag;

    // Set colors and initialize thread.
    public void init() {
        setBackground(Color.cyan);
        setForeground(Color.red);
    }

    // Start thread
    public void start() {
        msg = getParameter("message");
        if(msg == null) msg = "Message not found.";
        msg = " " + msg;
        t = new Thread(this);
        stopFlag = false;
        t.start();
    }

    // Entry point for the thread that runs the banner.
    public void run() {

        // Redisplay banner
        for( ; ; ) {
            try {
                repaint();
                Thread.sleep(250);
                if(stopFlag)
                    break;
            } catch(InterruptedException e) {}
        }
    }
}
```

```

// Pause the banner.
public void stop() {
    stopFlag = true;
    t = null;
}

// Display the banner.
public void paint(Graphics g) {
    char ch;

    ch = msg.charAt(0);
    msg = msg.substring(1, msg.length());
    msg += ch;

    g.drawString(msg, 50, 30);
}
}

```

getDocumentBase() and getCodeBase()

Often, you will create applets that will need to explicitly load media and text. Java will allow the applet to load data from the directory holding the HTML file that started the applet (the *document base*) and the directory from which the applet's class file was loaded (the *code base*). These directories are returned as **URL** objects (described in Chapter 22) by **getDocumentBase()** and **getCodeBase()**. They can be concatenated with a string that names the file you want to load. To actually load another file, you will use the **showDocument()** method defined by the **AppletContext** interface, discussed in the next section.

The following applet illustrates these methods:

```

// Display code and document bases.
import java.awt.*;
import java.applet.*;
import java.net.*;
/*
<applet code="Bases" width=300 height=50>
</applet>
*/

public class Bases extends Applet {
    // Display code and document bases.
    public void paint(Graphics g) {
        String msg;

        URL url = getCodeBase(); // get code base
        msg = "Code base: " + url.toString();
        g.drawString(msg, 10, 20);

        url = getDocumentBase(); // get document base
        msg = "Document base: " + url.toString();
        g.drawString(msg, 10, 40);
    }
}

```

Sample output from this program is shown here:



AppletContext and showDocument()

One application of Java is to use active images and animation to provide a graphical means of navigating the Web that is more interesting than simple text-based links. To allow your applet to transfer control to another URL, you must use the `showDocument()` method defined by the `AppletContext` interface. `AppletContext` is an interface that lets you get information from the applet's execution environment. The methods defined by `AppletContext` are shown in Table 23-2. The context of the currently executing applet is obtained by a call to the `getAppletContext()` method defined by `Applet`.

Method	Description
<code>Applet getApplet(String <i>appName</i>)</code>	Returns the applet specified by <i>appName</i> if it is within the current applet context. Otherwise, <code>null</code> is returned.
<code>Enumeration<Applet> getApplets()</code>	Returns an enumeration that contains all of the applets within the current applet context.
<code>AudioClip getAudioClip(URL <i>url</i>)</code>	Returns an <code>AudioClip</code> object that encapsulates the audio clip found at the location specified by <i>url</i> .
<code>Image getImage(URL <i>url</i>)</code>	Returns an <code>Image</code> object that encapsulates the image found at the location specified by <i>url</i> .
<code>InputStream getStream(String <i>key</i>)</code>	Returns the stream linked to <i>key</i> . Keys are linked to streams by using the <code>setStream()</code> method. A <code>null</code> reference is returned if no stream is linked to <i>key</i> .
<code>Iterator<String> getStreamKeys()</code>	Returns an iterator for the keys associated with the invoking object. The keys are linked to streams. See <code>getStream()</code> and <code>setStream()</code> .
<code>void setStream(String <i>key</i>, InputStream <i>strm</i>) throws IOException</code>	Links the stream specified by <i>strm</i> to the key passed in <i>key</i> . The <i>key</i> is deleted from the invoking object if <i>strm</i> is <code>null</code> .
<code>void showDocument(URL <i>url</i>)</code>	Brings the document at the <code>URL</code> specified by <i>url</i> into view. This method may not be supported by applet viewers.

Table 23-2 The Methods Defined by the `AppletContext` Interface

Method	Description
void showDocument(URL <i>url</i> , String <i>where</i>)	Brings the document at the URL specified by <i>url</i> into view. This method may not be supported by applet viewers. The placement of the document is specified by <i>where</i> as described in the text.
void showStatus(String <i>str</i>)	Displays <i>str</i> in the status window.

Table 23-2 The Methods Defined by the **AppletContext** Interface (*continued*)

Within an applet, once you have obtained the applet's context, you can bring another document into view by calling **showDocument()**. This method has no return value and throws no exception if it fails, so use it carefully. There are two **showDocument()** methods. The method **showDocument(URL)** displays the document at the specified **URL**. The method **showDocument(URL, String)** displays the specified document at the specified location within the browser window. Valid arguments for *where* are "*_self*" (show in current frame), "*_parent*" (show in parent frame), "*_top*" (show in topmost frame), and "*_blank*" (show in new browser window). You can also specify a name, which causes the document to be shown in a new browser window by that name.

The following applet demonstrates **AppletContext** and **showDocument()**. Upon execution, it obtains the current applet context and uses that context to transfer control to a file called **Test.html**. This file must be in the same directory as the applet. **Test.html** can contain any valid hypertext that you like.

```
/* Using an applet context, getCodeBase(),
   and showDocument() to display an HTML file.
*/

import java.awt.*;
import java.applet.*;
import java.net.*;
/*
<applet code="ACDemo" width=300 height=50>
</applet>
*/

public class ACDemo extends Applet {
    public void start() {
        AppletContext ac = getAppletContext();
        URL url = getCodeBase(); // get url of this applet

        try {
            ac.showDocument(new URL(url+"Test.html"));
        } catch(MalformedURLException e) {
            showStatus("URL not found");
        }
    }
}
```

The AudioClip Interface

The **AudioClip** interface defines these methods: **play()** (play a clip from the beginning), **stop()** (stop playing the clip), and **loop()** (play the loop continuously). After you have loaded an audio clip using **getAudioClip()**, you can use these methods to play it.

The AppletStub Interface

The **AppletStub** interface provides the means by which an applet and the browser (or applet viewer) communicate. Your code will not typically implement this interface.

Outputting to the Console

Although output to an applet's window must be accomplished through GUI-based methods, such as **drawString()**, it is still possible to use console output in your applet—especially for debugging purposes. In an applet, when you call a method such as **System.out.println()**, the output is not sent to your applet's window. Instead, it appears either in the console session in which you launched the applet viewer or in the Java console that is available in some browsers. Use of console output for purposes other than debugging is discouraged, since it violates the design principles of the graphical interface most users will expect.

This page has been intentionally left blank

CHAPTER

24

Event Handling

This chapter examines an important aspect of Java: the event. Event handling is fundamental to Java programming because it is integral to the creation of many kinds of applications, including applets and other types of GUI-based programs. As explained in Chapter 23, applets are event-driven programs that use a graphical user interface to interact with the user. Furthermore, any program that uses a graphical user interface, such as a Java application written for Windows, is event driven. Thus, you cannot write these types of programs without a solid command of event handling. Events are supported by a number of packages, including `java.util`, `java.awt`, and `java.awt.event`.

Most events to which your program will respond are generated when the user interacts with a GUI-based program. These are the types of events examined in this chapter. They are passed to your program in a variety of ways, with the specific method dependent upon the actual event. There are several types of events, including those generated by the mouse, the keyboard, and various GUI controls, such as a push button, scroll bar, or check box.

This chapter begins with an overview of Java's event handling mechanism. It then examines the main event classes and interfaces used by the AWT and develops several examples that demonstrate the fundamentals of event processing. This chapter also explains how to use adapter classes, inner classes, and anonymous inner classes to streamline event handling code. The examples provided in the remainder of this book make frequent use of these techniques.

NOTE This chapter focuses on events related to GUI-based programs. However, events are also occasionally used for purposes not directly related to GUI-based programs. In all cases, the same basic event handling techniques apply.

Two Event Handling Mechanisms

Before beginning our discussion of event handling, an important historical point must be made: The way in which events are handled changed significantly between the original version of Java (1.0) and all subsequent versions of Java, beginning with version 1.1.

Although the 1.0 method of event handling is still supported, it is not recommended for new programs. Also, many of the methods that support the old 1.0 event model have been deprecated. The modern approach is the way that events should be handled by all new programs and thus is the method employed by programs in this book.

The Delegation Event Model

The modern approach to handling events is based on the *delegation event model*, which defines standard and consistent mechanisms to generate and process events. Its concept is quite simple: a *source* generates an event and sends it to one or more *listeners*. In this scheme, the listener simply waits until it receives an event. Once an event is received, the listener processes the event and then returns. The advantage of this design is that the application logic that processes events is cleanly separated from the user interface logic that generates those events. A user interface element is able to “delegate” the processing of an event to a separate piece of code.

In the delegation event model, listeners must register with a source in order to receive an event notification. This provides an important benefit: notifications are sent only to listeners that want to receive them. This is a more efficient way to handle events than the design used by the original Java 1.0 approach. Previously, an event was propagated up the containment hierarchy until it was handled by a component. This required components to receive events that they did not process, and it wasted valuable time. The delegation event model eliminates this overhead.

The following sections define events and describe the roles of sources and listeners.

Events

In the delegation model, an *event* is an object that describes a state change in a source. Among other causes, an event can be generated as a consequence of a person interacting with the elements in a graphical user interface. Some of the activities that cause events to be generated are pressing a button, entering a character via the keyboard, selecting an item in a list, and clicking the mouse. Many other user operations could also be cited as examples.

Events may also occur that are not directly caused by interactions with a user interface. For example, an event may be generated when a timer expires, a counter exceeds a value, a software or hardware failure occurs, or an operation is completed. You are free to define events that are appropriate for your application.

Event Sources

A *source* is an object that generates an event. This occurs when the internal state of that object changes in some way. Sources may generate more than one type of event.

A source must register listeners in order for the listeners to receive notifications about a specific type of event. Each type of event has its own registration method. Here is the general form:

```
public void addTypeListener (TypeListener el)
```

Here, *Type* is the name of the event, and *el* is a reference to the event listener. For example, the method that registers a keyboard event listener is called **addKeyListener()**. The method that registers a mouse motion listener is called **addMouseMotionListener()**. When an event

occurs, all registered listeners are notified and receive a copy of the event object. This is known as *multicasting* the event. In all cases, notifications are sent only to listeners that register to receive them.

Some sources may allow only one listener to register. The general form of such a method is this:

```
public void addTypeListener(TypeListener el)
    throws java.util.TooManyListenersException
```

Here, *Type* is the name of the event, and *el* is a reference to the event listener. When such an event occurs, the registered listener is notified. This is known as *unicasting* the event.

A source must also provide a method that allows a listener to unregister an interest in a specific type of event. The general form of such a method is this:

```
public void removeTypeListener(TypeListener el)
```

Here, *Type* is the name of the event, and *el* is a reference to the event listener. For example, to remove a keyboard listener, you would call `removeKeyListener()`.

The methods that add or remove listeners are provided by the source that generates events. For example, the **Component** class provides methods to add and remove keyboard and mouse event listeners.

Event Listeners

A *listener* is an object that is notified when an event occurs. It has two major requirements. First, it must have been registered with one or more sources to receive notifications about specific types of events. Second, it must implement methods to receive and process these notifications.

The methods that receive and process events are defined in a set of interfaces, such as those found in **java.awt.event**. For example, the **MouseMotionListener** interface defines two methods to receive notifications when the mouse is dragged or moved. Any object may receive and process one or both of these events if it provides an implementation of this interface. Other listener interfaces are discussed later in this and other chapters.

Event Classes

The classes that represent events are at the core of Java's event handling mechanism. Thus, a discussion of event handling must begin with the event classes. It is important to understand, however, that Java defines several types of events and that not all event classes can be discussed in this chapter. Arguably, the most widely used events at the time of this writing are those defined by the AWT and those defined by Swing. This chapter focuses on the AWT events. (Most of these events also apply to Swing.) Several Swing-specific events are described in Chapter 31, when Swing is covered.

At the root of the Java event class hierarchy is **EventObject**, which is in **java.util**. It is the superclass for all events. Its one constructor is shown here:

```
EventObject(Object src)
```

Here, *src* is the object that generates this event.

EventObject defines two methods: **getSource()** and **toString()**. The **getSource()** method returns the source of the event. Its general form is shown here:

```
Object getSource()
```

As expected, **toString()** returns the string equivalent of the event.

The class **AWTEvent**, defined within the **java.awt** package, is a subclass of **EventObject**. It is the superclass (either directly or indirectly) of all AWT-based events used by the delegation event model. Its **getID()** method can be used to determine the type of the event. The signature of this method is shown here:

```
int getID()
```

Additional details about **AWTEvent** are provided at the end of Chapter 26. At this point, it is important to know only that all of the other classes discussed in this section are subclasses of **AWTEvent**.

To summarize:

- **EventObject** is a superclass of all events.
- **AWTEvent** is a superclass of all AWT events that are handled by the delegation event model.

The package **java.awt.event** defines many types of events that are generated by various user interface elements. Table 24-1 shows several commonly used event classes and provides a brief description of when they are generated. Commonly used constructors and methods in each class are described in the following sections.

Event Class	Description
ActionEvent	Generated when a button is pressed, a list item is double-clicked, or a menu item is selected.
AdjustmentEvent	Generated when a scroll bar is manipulated.
ComponentEvent	Generated when a component is hidden, moved, resized, or becomes visible.
ContainerEvent	Generated when a component is added to or removed from a container.
FocusEvent	Generated when a component gains or loses keyboard focus.
InputEvent	Abstract superclass for all component input event classes.
ItemEvent	Generated when a check box or list item is clicked; also occurs when a choice selection is made or a checkable menu item is selected or deselected.
KeyEvent	Generated when input is received from the keyboard.
MouseEvent	Generated when the mouse is dragged, moved, clicked, pressed, or released; also generated when the mouse enters or exits a component.
MouseWheelEvent	Generated when the mouse wheel is moved.
TextEvent	Generated when the value of a text area or text field is changed.
WindowEvent	Generated when a window is activated, closed, deactivated, deiconified, iconified, opened, or quit.

Table 24-1 Commonly Used Event Classes in **java.awt.event**

The ActionEvent Class

An **ActionEvent** is generated when a button is pressed, a list item is double-clicked, or a menu item is selected. The **ActionEvent** class defines four integer constants that can be used to identify any modifiers associated with an action event: **ALT_MASK**, **CTRL_MASK**, **META_MASK**, and **SHIFT_MASK**. In addition, there is an integer constant, **ACTION_PERFORMED**, which can be used to identify action events.

ActionEvent has these three constructors:

```
ActionEvent(Object src, int type, String cmd)
ActionEvent(Object src, int type, String cmd, int modifiers)
ActionEvent(Object src, int type, String cmd, long when, int modifiers)
```

Here, *src* is a reference to the object that generated this event. The type of the event is specified by *type*, and its command string is *cmd*. The argument *modifiers* indicates which modifier keys (ALT, CTRL, META, and/or SHIFT) were pressed when the event was generated. The *when* parameter specifies when the event occurred.

You can obtain the command name for the invoking **ActionEvent** object by using the **getActionCommand()** method, shown here:

```
String getActionCommand()
```

For example, when a button is pressed, an action event is generated that has a command name equal to the label on that button.

The **getModifiers()** method returns a value that indicates which modifier keys (ALT, CTRL, META, and/or SHIFT) were pressed when the event was generated. Its form is shown here:

```
int getModifiers()
```

The method **getWhen()** returns the time at which the event took place. This is called the event's *timestamp*. The **getWhen()** method is shown here:

```
long getWhen()
```

The AdjustmentEvent Class

An **AdjustmentEvent** is generated by a scroll bar. There are five types of adjustment events. The **AdjustmentEvent** class defines integer constants that can be used to identify them. The constants and their meanings are shown here:

BLOCK_DECREMENT	The user clicked inside the scroll bar to decrease its value.
BLOCK_INCREMENT	The user clicked inside the scroll bar to increase its value.
TRACK	The slider was dragged.
UNIT_DECREMENT	The button at the end of the scroll bar was clicked to decrease its value.
UNIT_INCREMENT	The button at the end of the scroll bar was clicked to increase its value.

In addition, there is an integer constant, **ADJUSTMENT_VALUE_CHANGED**, that indicates that a change has occurred.

Here is one **AdjustmentEvent** constructor:

```
AdjustmentEvent(Adjustable src, int id, int type, int val)
```

Here, *src* is a reference to the object that generated this event. The *id* specifies the event. The type of the adjustment is specified by *type*, and its associated value is *val*.

The **getAdjustable()** method returns the object that generated the event. Its form is shown here:

```
Adjustable getAdjustable()
```

The type of the adjustment event may be obtained by the **getAdjustmentType()** method. It returns one of the constants defined by **AdjustmentEvent**. The general form is shown here:

```
int getAdjustmentType()
```

The amount of the adjustment can be obtained from the **getValue()** method, shown here:

```
int getValue()
```

For example, when a scroll bar is manipulated, this method returns the value represented by that change.

The ComponentEvent Class

A **ComponentEvent** is generated when the size, position, or visibility of a component is changed. There are four types of component events. The **ComponentEvent** class defines integer constants that can be used to identify them. The constants and their meanings are shown here:

COMPONENT_HIDDEN	The component was hidden.
COMPONENT_MOVED	The component was moved.
COMPONENT_RESIZED	The component was resized.
COMPONENT_SHOWN	The component became visible.

ComponentEvent has this constructor:

```
ComponentEvent(Component src, int type)
```

Here, *src* is a reference to the object that generated this event. The type of the event is specified by *type*.

ComponentEvent is the superclass either directly or indirectly of **ContainerEvent**, **FocusEvent**, **KeyEvent**, **MouseEvent**, and **WindowEvent**, among others.

The **getComponent()** method returns the component that generated the event. Its form is shown here:

```
Component getComponent()
```

The ContainerEvent Class

A **ContainerEvent** is generated when a component is added to or removed from a container. There are two types of container events. The **ContainerEvent** class defines integer constants that can be used to identify them: **COMPONENT_ADDED** and

COMPONENT_REMOVED. They indicate that a component has been added to or removed from the container.

ContainerEvent is a subclass of **ComponentEvent** and has this constructor:

```
ContainerEvent(Component src, int type, Component comp)
```

Here, *src* is a reference to the container that generated this event. The type of the event is specified by *type*, and the component that has been added to or removed from the container is *comp*.

You can obtain a reference to the container that generated this event by using the **getContainer()** method, shown here:

```
Container getContainer()
```

The **getChild()** method returns a reference to the component that was added to or removed from the container. Its general form is shown here:

```
Component getChild()
```

The FocusEvent Class

A **FocusEvent** is generated when a component gains or loses input focus. These events are identified by the integer constants **FOCUS_GAINED** and **FOCUS_LOST**.

FocusEvent is a subclass of **ComponentEvent** and has these constructors:

```
FocusEvent(Component src, int type)
```

```
FocusEvent(Component src, int type, boolean temporaryFlag)
```

```
FocusEvent(Component src, int type, boolean temporaryFlag, Component other)
```

Here, *src* is a reference to the component that generated this event. The type of the event is specified by *type*. The argument *temporaryFlag* is set to **true** if the focus event is temporary. Otherwise, it is set to **false**. (A temporary focus event occurs as a result of another user interface operation. For example, assume that the focus is in a text field. If the user moves the mouse to adjust a scroll bar, the focus is temporarily lost.)

The other component involved in the focus change, called the *opposite component*, is passed in *other*. Therefore, if a **FOCUS_GAINED** event occurred, *other* will refer to the component that lost focus. Conversely, if a **FOCUS_LOST** event occurred, *other* will refer to the component that gains focus.

You can determine the other component by calling **getOppositeComponent()**, shown here:

```
Component getOppositeComponent()
```

The opposite component is returned.

The **isTemporary()** method indicates if this focus change is temporary. Its form is shown here:

```
boolean isTemporary()
```

The method returns **true** if the change is temporary. Otherwise, it returns **false**.

The InputEvent Class

The abstract class **InputEvent** is a subclass of **ComponentEvent** and is the superclass for component input events. Its subclasses are **KeyEvent** and **MouseEvent**.

InputEvent defines several integer constants that represent any modifiers, such as the control key being pressed, that might be associated with the event. Originally, the **InputEvent** class defined the following eight values to represent the modifiers:

ALT_MASK	BUTTON2_MASK	META_MASK
ALT_GRAPH_MASK	BUTTON3_MASK	SHIFT_MASK
BUTTON1_MASK	CTRL_MASK	

However, because of possible conflicts between the modifiers used by keyboard events and mouse events, and other issues, the following extended modifier values were added:

ALT_DOWN_MASK	BUTTON2_DOWN_MASK	META_DOWN_MASK
ALT_GRAPH_DOWN_MASK	BUTTON3_DOWN_MASK	SHIFT_DOWN_MASK
BUTTON1_DOWN_MASK	CTRL_DOWN_MASK	

When writing new code, it is recommended that you use the new, extended modifiers rather than the original modifiers.

To test if a modifier was pressed at the time an event is generated, use the **isAltDown()**, **isAltGraphDown()**, **isControlDown()**, **isMetaDown()**, and **isShiftDown()** methods. The forms of these methods are shown here:

```
boolean isAltDown()
boolean isAltGraphDown()
boolean isControlDown()
boolean isMetaDown()
boolean isShiftDown()
```

You can obtain a value that contains all of the original modifier flags by calling the **getModifiers()** method. It is shown here:

```
int getModifiers()
```

You can obtain the extended modifiers by calling **getModifiersEx()**, which is shown here:

```
int getModifiersEx()
```

The ItemEvent Class

An **ItemEvent** is generated when a check box or a list item is clicked or when a checkable menu item is selected or deselected. (Check boxes and list boxes are described later in this book.) There are two types of item events, which are identified by the following integer constants:

DESELECTED	The user deselected an item.
SELECTED	The user selected an item.

In addition, **ItemEvent** defines one integer constant, **ITEM_STATE_CHANGED**, that signifies a change of state.

ItemEvent has this constructor:

```
ItemEvent(ItemSelectable src, int type, Object entry, int state)
```

Here, *src* is a reference to the component that generated this event. For example, this might be a list or choice element. The type of the event is specified by *type*. The specific item that generated the item event is passed in *entry*. The current state of that item is in *state*.

The **getItem()** method can be used to obtain a reference to the item that changed. Its signature is shown here:

```
Object getItem()
```

The **getItemSelectable()** method can be used to obtain a reference to the **ItemSelectable** object that generated an event. Its general form is shown here:

```
ItemSelectable getItemSelectable()
```

Lists and choices are examples of user interface elements that implement the **ItemSelectable** interface.

The **getStateChange()** method returns the state change (that is, **SELECTED** or **DESELECTED**) for the event. It is shown here:

```
int getStateChange()
```

The KeyEvent Class

A **KeyEvent** is generated when keyboard input occurs. There are three types of key events, which are identified by these integer constants: **KEY_PRESSED**, **KEY_RELEASED**, and **KEY_TYPED**. The first two events are generated when any key is pressed or released. The last event occurs only when a character is generated. Remember, not all keypresses result in characters. For example, pressing SHIFT does not generate a character.

There are many other integer constants that are defined by **KeyEvent**. For example, **VK_0** through **VK_9** and **VK_A** through **VK_Z** define the ASCII equivalents of the numbers and letters. Here are some others:

VK_ALT	VK_DOWN	VK_LEFT	VK_RIGHT
VK_CANCEL	VK_ENTER	VK_PAGE_DOWN	VK_SHIFT
VK_CONTROL	VK_ESCAPE	VK_PAGE_UP	VK_UP

The **VK** constants specify *virtual key codes* and are independent of any modifiers, such as control, shift, or alt.

KeyEvent is a subclass of **InputEvent**. Here is one of its constructors:

```
KeyEvent(Component src, int type, long when, int modifiers, int code, char ch)
```

Here, *src* is a reference to the component that generated the event. The type of the event is specified by *type*. The system time at which the key was pressed is passed in *when*. The

modifiers argument indicates which modifiers were pressed when this key event occurred. The virtual key code, such as `VK_UP`, `VK_A`, and so forth, is passed in *code*. The character equivalent (if one exists) is passed in *ch*. If no valid character exists, then *ch* contains `CHAR_UNDEFINED`. For `KEY_TYPED` events, *code* will contain `VK_UNDEFINED`.

The `KeyEvent` class defines several methods, but probably the most commonly used ones are `getKeyChar()`, which returns the character that was entered, and `getKeyCode()`, which returns the key code. Their general forms are shown here:

```
char getKeyChar()
int getKeyCode()
```

If no valid character is available, then `getKeyChar()` returns `CHAR_UNDEFINED`. When a `KEY_TYPED` event occurs, `getKeyCode()` returns `VK_UNDEFINED`.

The MouseEvent Class

There are eight types of mouse events. The `MouseEvent` class defines the following integer constants that can be used to identify them:

MOUSE_CLICKED	The user clicked the mouse.
MOUSE_DRAGGED	The user dragged the mouse.
MOUSE_ENTERED	The mouse entered a component.
MOUSE_EXITED	The mouse exited from a component.
MOUSE_MOVED	The mouse moved.
MOUSE_PRESSED	The mouse was pressed.
MOUSE_RELEASED	The mouse was released.
MOUSE_WHEEL	The mouse wheel was moved.

`MouseEvent` is a subclass of `InputEvent`. Here is one of its constructors:

```
MouseEvent(Component src, int type, long when, int modifiers,
           int x, int y, int clicks, boolean triggersPopup)
```

Here, *src* is a reference to the component that generated the event. The type of the event is specified by *type*. The system time at which the mouse event occurred is passed in *when*. The *modifiers* argument indicates which modifiers were pressed when a mouse event occurred. The coordinates of the mouse are passed in *x* and *y*. The click count is passed in *clicks*. The *triggersPopup* flag indicates if this event causes a pop-up menu to appear on this platform.

Two commonly used methods in this class are `getX()` and `getY()`. These return the X and Y coordinates of the mouse within the component when the event occurred. Their forms are shown here:

```
int getX()
int getY()
```

Alternatively, you can use the `getPoint()` method to obtain the coordinates of the mouse. It is shown here:

```
Point getPoint()
```

It returns a `Point` object that contains the X,Y coordinates in its integer members: `x` and `y`.

The **translatePoint()** method changes the location of the event. Its form is shown here:

```
void translatePoint(int x, int y)
```

Here, the arguments *x* and *y* are added to the coordinates of the event.

The **getClickCount()** method obtains the number of mouse clicks for this event. Its signature is shown here:

```
int getClickCount()
```

The **isPopupTrigger()** method tests if this event causes a pop-up menu to appear on this platform. Its form is shown here:

```
boolean isPopupTrigger()
```

Also available is the **getButton()** method, shown here:

```
int getButton()
```

It returns a value that represents the button that caused the event. For most cases, the return value will be one of these constants defined by **MouseEvent**:

NOBUTTON	BUTTON1	BUTTON2	BUTTON3
----------	---------	---------	---------

The **NOBUTTON** value indicates that no button was pressed or released.

Also available are three methods that obtain the coordinates of the mouse relative to the screen rather than the component. They are shown here:

```
Point getLocationOnScreen()
```

```
int getXOnScreen()
```

```
int getYOnScreen()
```

The **getLocationOnScreen()** method returns a **Point** object that contains both the X and Y coordinate. The other two methods return the indicated coordinate.

The MouseWheelEvent Class

The **MouseWheelEvent** class encapsulates a mouse wheel event. It is a subclass of **MouseEvent**. Not all mice have wheels. If a mouse has a wheel, it is typically located between the left and right buttons. Mouse wheels are used for scrolling. **MouseWheelEvent** defines these two integer constants:

WHEEL_BLOCK_SCROLL	A page-up or page-down scroll event occurred.
WHEEL_UNIT_SCROLL	A line-up or line-down scroll event occurred.

Here is one of the constructors defined by **MouseWheelEvent**:

```
MouseWheelEvent(Component src, int type, long when, int modifiers,
                int x, int y, int clicks, boolean triggersPopup,
                int scrollHow, int amount, int count)
```

Here, *src* is a reference to the object that generated the event. The type of the event is specified by *type*. The system time at which the mouse event occurred is passed in *when*.

The *modifiers* argument indicates which modifiers were pressed when the event occurred. The coordinates of the mouse are passed in *x* and *y*. The number of clicks is passed in *clicks*. The *triggersPopup* flag indicates if this event causes a pop-up menu to appear on this platform. The *scrollHow* value must be either **WHEEL_UNIT_SCROLL** or **WHEEL_BLOCK_SCROLL**. The number of units to scroll is passed in *amount*. The *count* parameter indicates the number of rotational units that the wheel moved.

MouseWheelEvent defines methods that give you access to the wheel event. To obtain the number of rotational units, call **getWheelRotation()**, shown here:

```
int getWheelRotation()
```

It returns the number of rotational units. If the value is positive, the wheel moved counterclockwise. If the value is negative, the wheel moved clockwise. JDK 7 added a method called **getPreciseWheelRotation()**, which supports high-resolution wheels. It works like **getWheelRotation()**, but returns a **double**.

To obtain the type of scroll, call **getScrollType()**, shown next:

```
int getScrollType()
```

It returns either **WHEEL_UNIT_SCROLL** or **WHEEL_BLOCK_SCROLL**.

If the scroll type is **WHEEL_UNIT_SCROLL**, you can obtain the number of units to scroll by calling **getScrollAmount()**. It is shown here:

```
int getScrollAmount()
```

The TextEvent Class

Instances of this class describe text events. These are generated by text fields and text areas when characters are entered by a user or program. **TextEvent** defines the integer constant **TEXT_VALUE_CHANGED**.

The one constructor for this class is shown here:

```
TextEvent(Object src, int type)
```

Here, *src* is a reference to the object that generated this event. The type of the event is specified by *type*.

The **TextEvent** object does not include the characters currently in the text component that generated the event. Instead, your program must use other methods associated with the text component to retrieve that information. This operation differs from other event objects discussed in this section. Think of a text event notification as a signal to a listener that it should retrieve information from a specific text component.

The WindowEvent Class

There are ten types of window events. The **WindowEvent** class defines integer constants that can be used to identify them. The constants and their meanings are shown here:

WINDOW_ACTIVATED	The window was activated.
WINDOW_CLOSED	The window has been closed.

WINDOW_CLOSING	The user requested that the window be closed.
WINDOW_DEACTIVATED	The window was deactivated.
WINDOW_DEICONIFIED	The window was deiconified.
WINDOW_GAINED_FOCUS	The window gained input focus.
WINDOW_ICONIFIED	The window was iconified.
WINDOW_LOST_FOCUS	The window lost input focus.
WINDOW_OPENED	The window was opened.
WINDOW_STATE_CHANGED	The state of the window changed.

WindowEvent is a subclass of **ComponentEvent**. It defines several constructors. The first is

`WindowEvent(Window src, int type)`

Here, *src* is a reference to the object that generated this event. The type of the event is *type*.

The next three constructors offer more detailed control:

`WindowEvent(Window src, int type, Window other)`

`WindowEvent(Window src, int type, int fromState, int toState)`

`WindowEvent(Window src, int type, Window other, int fromState, int toState)`

Here, *other* specifies the opposite window when a focus or activation event occurs. The *fromState* specifies the prior state of the window, and *toState* specifies the new state that the window will have when a window state change occurs.

A commonly used method in this class is `getWindow()`. It returns the **Window** object that generated the event. Its general form is shown here:

`Window getWindow()`

WindowEvent also defines methods that return the opposite window (when a focus or activation event has occurred), the previous window state, and the current window state. These methods are shown here:

```
Window getOppositeWindow()
int getOldState()
int getNewState()
```

Sources of Events

Table 24-2 lists some of the user interface components that can generate the events described in the previous section. In addition to these graphical user interface elements, any class derived from **Component**, such as **Applet**, can generate events. For example, you can receive key and mouse events from an applet. (You may also build your own components that generate events.) In this chapter, we will be handling only mouse and keyboard events, but the following two chapters will be handling events from the sources shown in Table 24-2.

Event Source	Description
Button	Generates action events when the button is pressed.
Check box	Generates item events when the check box is selected or deselected.
Choice	Generates item events when the choice is changed.
List	Generates action events when an item is double-clicked; generates item events when an item is selected or deselected.
Menu item	Generates action events when a menu item is selected; generates item events when a checkable menu item is selected or deselected.
Scroll bar	Generates adjustment events when the scroll bar is manipulated.
Text components	Generates text events when the user enters a character.
Window	Generates window events when a window is activated, closed, deactivated, deiconified, iconified, opened, or quit.

Table 24-2 Event Source Examples

Event Listener Interfaces

As explained, the delegation event model has two parts: sources and listeners. As it relates to this chapter, listeners are created by implementing one or more of the interfaces defined by the `java.awt.event` package. When an event occurs, the event source invokes the appropriate method defined by the listener and provides an event object as its argument. Table 24-3 lists

Interface	Description
ActionListener	Defines one method to receive action events.
AdjustmentListener	Defines one method to receive adjustment events.
ComponentListener	Defines four methods to recognize when a component is hidden, moved, resized, or shown.
ContainerListener	Defines two methods to recognize when a component is added to or removed from a container.
FocusListener	Defines two methods to recognize when a component gains or loses keyboard focus.
ItemListener	Defines one method to recognize when the state of an item changes.
KeyListener	Defines three methods to recognize when a key is pressed, released, or typed.
MouseListener	Defines five methods to recognize when the mouse is clicked, enters a component, exits a component, is pressed, or is released.
MouseMotionListener	Defines two methods to recognize when the mouse is dragged or moved.
MouseWheelListener	Defines one method to recognize when the mouse wheel is moved.
TextListener	Defines one method to recognize when a text value changes.
WindowFocusListener	Defines two methods to recognize when a window gains or loses input focus.
WindowListener	Defines seven methods to recognize when a window is activated, closed, deactivated, deiconified, iconified, opened, or quit.

Table 24-3 Commonly Used Event Listener Interfaces

several commonly used listener interfaces and provides a brief description of the methods that they define. The following sections examine the specific methods that are contained in each interface.

The ActionListener Interface

This interface defines the **actionPerformed()** method that is invoked when an action event occurs. Its general form is shown here:

```
void actionPerformed(ActionEvent ae)
```

The AdjustmentListener Interface

This interface defines the **adjustmentValueChanged()** method that is invoked when an adjustment event occurs. Its general form is shown here:

```
void adjustmentValueChanged(AdjustmentEvent ae)
```

The ComponentListener Interface

This interface defines four methods that are invoked when a component is resized, moved, shown, or hidden. Their general forms are shown here:

```
void componentResized(ComponentEvent ce)
void componentMoved(ComponentEvent ce)
void componentShown(ComponentEvent ce)
void componentHidden(ComponentEvent ce)
```

The ContainerListener Interface

This interface contains two methods. When a component is added to a container, **componentAdded()** is invoked. When a component is removed from a container, **componentRemoved()** is invoked. Their general forms are shown here:

```
void componentAdded(ContainerEvent ce)
void componentRemoved(ContainerEvent ce)
```

The FocusListener Interface

This interface defines two methods. When a component obtains keyboard focus, **focusGained()** is invoked. When a component loses keyboard focus, **focusLost()** is called. Their general forms are shown here:

```
void focusGained(FocusEvent fe)
void focusLost(FocusEvent fe)
```

The ItemListener Interface

This interface defines the **itemStateChanged()** method that is invoked when the state of an item changes. Its general form is shown here:

```
void itemStateChanged(ItemEvent ie)
```

The KeyListener Interface

This interface defines three methods. The **keyPressed()** and **keyReleased()** methods are invoked when a key is pressed and released, respectively. The **keyTyped()** method is invoked when a character has been entered.

For example, if a user presses and releases the `A` key, three events are generated in sequence: key pressed, typed, and released. If a user presses and releases the `HOME` key, two key events are generated in sequence: key pressed and released.

The general forms of these methods are shown here:

```
void keyPressed(KeyEvent ke)
void keyReleased(KeyEvent ke)
void keyTyped(KeyEvent ke)
```

The MouseListener Interface

This interface defines five methods. If the mouse is pressed and released at the same point, **mouseClicked()** is invoked. When the mouse enters a component, the **mouseEntered()** method is called. When it leaves, **mouseExited()** is called. The **mousePressed()** and **mouseReleased()** methods are invoked when the mouse is pressed and released, respectively.

The general forms of these methods are shown here:

```
void mouseClicked(MouseEvent me)
void mouseEntered(MouseEvent me)
void mouseExited(MouseEvent me)
void mousePressed(MouseEvent me)
void mouseReleased(MouseEvent me)
```

The MouseMotionListener Interface

This interface defines two methods. The **mouseDragged()** method is called multiple times as the mouse is dragged. The **mouseMoved()** method is called multiple times as the mouse is moved. Their general forms are shown here:

```
void mouseDragged(MouseEvent me)
void mouseMoved(MouseEvent me)
```

The MouseWheelListener Interface

This interface defines the **mouseWheelMoved()** method that is invoked when the mouse wheel is moved. Its general form is shown here:

```
void mouseWheelMoved(MouseWheelEvent mwe)
```

The TextListener Interface

This interface defines the **textValueChanged()** method that is invoked when a change occurs in a text area or text field. Its general form is shown here:

```
void textValueChanged(TextEvent te)
```

The WindowFocusListener Interface

This interface defines two methods: `windowGainedFocus()` and `windowLostFocus()`. These are called when a window gains or loses input focus. Their general forms are shown here:

```
void windowGainedFocus(WindowEvent we)
void windowLostFocus(WindowEvent we)
```

The WindowListener Interface

This interface defines seven methods. The `windowActivated()` and `windowDeactivated()` methods are invoked when a window is activated or deactivated, respectively. If a window is iconified, the `windowIconified()` method is called. When a window is deiconified, the `windowDeiconified()` method is called. When a window is opened or closed, the `windowOpened()` or `windowClosed()` methods are called, respectively. The `windowClosing()` method is called when a window is being closed. The general forms of these methods are

```
void windowActivated(WindowEvent we)
void windowClosed(WindowEvent we)
void windowClosing(WindowEvent we)
void windowDeactivated(WindowEvent we)
void windowDeiconified(WindowEvent we)
void windowIconified(WindowEvent we)
void windowOpened(WindowEvent we)
```

Using the Delegation Event Model

Now that you have learned the theory behind the delegation event model and have had an overview of its various components, it is time to see it in practice. Using the delegation event model is actually quite easy. Just follow these two steps:

1. Implement the appropriate interface in the listener so that it can receive the type of event desired.
2. Implement code to register and unregister (if necessary) the listener as a recipient for the event notifications.

Remember that a source may generate several types of events. Each event must be registered separately. Also, an object may register to receive several types of events, but it must implement all of the interfaces that are required to receive these events.

To see how the delegation model works in practice, we will look at examples that handle two commonly used event generators: the mouse and keyboard.

Handling Mouse Events

To handle mouse events, you must implement the `MouseListener` and the `MouseMotionListener` interfaces. (You may also want to implement `MouseWheelListener`, but we won't be doing so, here.) The following applet demonstrates the process. It displays the current coordinates of the mouse in the applet's status window. Each time a button is

pressed, the word "Down" is displayed at the location of the mouse pointer. Each time the button is released, the word "Up" is shown. If a button is clicked, the message "Mouse clicked" is displayed in the upper-left corner of the applet display area.

As the mouse enters or exits the applet window, a message is displayed in the upper-left corner of the applet display area. When dragging the mouse, a * is shown, which tracks with the mouse pointer as it is dragged. Notice that the two variables, **mouseX** and **mouseY**, store the location of the mouse when a mouse pressed, released, or dragged event occurs. These coordinates are then used by **paint()** to display output at the point of these occurrences.

```
// Demonstrate the mouse event handlers.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="MouseEvents" width=300 height=100>
</applet>
*/
public class MouseEvents extends Applet
    implements MouseListener, MouseMotionListener {

    String msg = "";
    int mouseX = 0, mouseY = 0; // coordinates of mouse

    public void init() {
        addMouseListener(this);
        addMouseMotionListener(this);
    }

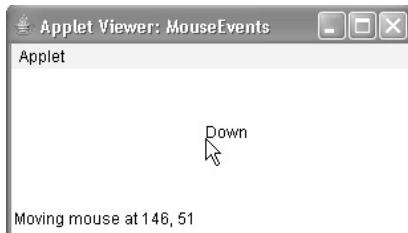
    // Handle mouse clicked.
    public void mouseClicked(MouseEvent me) {
        // save coordinates
        mouseX = 0;
        mouseY = 10;
        msg = "Mouse clicked.";
        repaint();
    }

    // Handle mouse entered.
    public void mouseEntered(MouseEvent me) {
        // save coordinates
        mouseX = 0;
        mouseY = 10;
        msg = "Mouse entered.";
        repaint();
    }

    // Handle mouse exited.
```

```
public void mouseExited(MouseEvent me) {  
    // save coordinates  
    mouseX = 0;  
    mouseY = 10;  
    msg = "Mouse exited.";  
    repaint();  
}  
  
// Handle button pressed.  
public void mousePressed(MouseEvent me) {  
    // save coordinates  
    mouseX = me.getX();  
    mouseY = me.getY();  
    msg = "Down";  
    repaint();  
}  
  
// Handle button released.  
public void mouseReleased(MouseEvent me) {  
    // save coordinates  
    mouseX = me.getX();  
    mouseY = me.getY();  
    msg = "Up";  
    repaint();  
}  
  
// Handle mouse dragged.  
public void mouseDragged(MouseEvent me) {  
    // save coordinates  
    mouseX = me.getX();  
    mouseY = me.getY();  
    msg = "*";  
    showStatus("Dragging mouse at " + mouseX + ", " + mouseY);  
    repaint();  
}  
  
// Handle mouse moved.  
public void mouseMoved(MouseEvent me) {  
    // show status  
    showStatus("Moving mouse at " + me.getX() + ", " + me.getY());  
}  
  
// Display msg in applet window at current X,Y location.  
public void paint(Graphics g) {  
    g.drawString(msg, mouseX, mouseY);  
}  
}
```

Sample output from this program is shown here:



Let's look closely at this example. The **MouseEvents** class extends **Applet** and implements both the **MouseListener** and **MouseMotionListener** interfaces. These two interfaces contain methods that receive and process the various types of mouse events. Notice that the applet is both the source and the listener for these events. This works because **Component**, which supplies the **addMouseListener()** and **addMouseMotionListener()** methods, is a superclass of **Applet**. Being both the source and the listener for events is a common situation for applets.

Inside **init()**, the applet registers itself as a listener for mouse events. This is done by using **addMouseListener()** and **addMouseMotionListener()**, which, as mentioned, are members of **Component**. They are shown here:

```
void addMouseListener(MouseListener ml)
void addMouseMotionListener(MouseMotionListener mml)
```

Here, *ml* is a reference to the object receiving mouse events, and *mml* is a reference to the object receiving mouse motion events. In this program, the same object is used for both.

The applet then implements all of the methods defined by the **MouseListener** and **MouseMotionListener** interfaces. These are the event handlers for the various mouse events. Each method handles its event and then returns.

Handling Keyboard Events

To handle keyboard events, you use the same general architecture as that shown in the mouse event example in the preceding section. The difference, of course, is that you will be implementing the **KeyListener** interface.

Before looking at an example, it is useful to review how key events are generated. When a key is pressed, a **KEY_PRESSED** event is generated. This results in a call to the **keyPressed()** event handler. When the key is released, a **KEY_RELEASED** event is generated and the **keyReleased()** handler is executed. If a character is generated by the keystroke, then a **KEY_TYPED** event is sent and the **keyTyped()** handler is invoked. Thus, each time the user presses a key, at least two and often three events are generated. If all you care about are actual characters, then you can ignore the information passed by the key press and release events. However, if your program needs to handle special keys, such as the arrow or function keys, then it must watch for them through the **keyPressed()** handler.

The following program demonstrates keyboard input. It echoes keystrokes to the applet window and shows the pressed/released status of each key in the status window.

```
// Demonstrate the key event handlers.
import java.awt.*;
import java.awt.event.*;
```

```
import java.applet.*;
/*
<applet code="SimpleKey" width=300 height=100>
</applet>
*/
public class SimpleKey extends Applet
    implements KeyListener {

    String msg = "";
    int X = 10, Y = 20; // output coordinates

    public void init() {
        addKeyListener(this);
    }

    public void keyPressed(KeyEvent ke) {
        showStatus("Key Down");
    }

    public void keyReleased(KeyEvent ke) {
        showStatus("Key Up");
    }

    public void keyTyped(KeyEvent ke) {
        msg += ke.getKeyChar();
        repaint();
    }

    // Display keystrokes.
    public void paint(Graphics g) {
        g.drawString(msg, X, Y);
    }
}
```

Sample output is shown here:



If you want to handle the special keys, such as the arrow or function keys, you need to respond to them within the **keyPressed()** handler. They are not available through **keyTyped()**.

To identify the keys, you use their virtual key codes. For example, the next applet outputs the name of a few of the special keys:

```
// Demonstrate some virtual key codes.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="KeyEvents" width=300 height=100>
</applet>
*/
public class KeyEvents extends Applet
implements KeyListener {

    String msg = "";
    int X = 10, Y = 20; // output coordinates

    public void init() {
        addKeyListener(this);
    }

    public void keyPressed(KeyEvent ke) {
        showStatus("Key Down");

        int key = ke.getKeyCode();
        switch(key) {
            case KeyEvent.VK_F1:
                msg += "<F1>";
                break;
            case KeyEvent.VK_F2:
                msg += "<F2>";
                break;
            case KeyEvent.VK_F3:
                msg += "<F3>";
                break;
            case KeyEvent.VK_PAGE_DOWN:
                msg += "<PgDn>";
                break;
            case KeyEvent.VK_PAGE_UP:
                msg += "<PgUp>";
                break;
            case KeyEvent.VK_LEFT:
                msg += "<Left Arrow>";
                break;
            case KeyEvent.VK_RIGHT:
                msg += "<Right Arrow>";
                break;
        }
        repaint();
    }
}
```

```

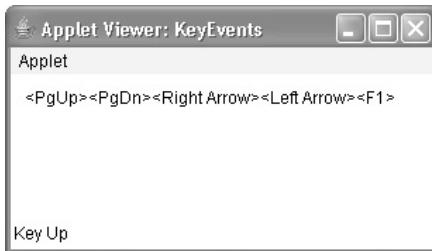
public void keyReleased(KeyEvent ke) {
    showStatus("Key Up");
}

public void keyTyped(KeyEvent ke) {
    msg += ke.getKeyChar();
    repaint();
}

// Display keystrokes.
public void paint(Graphics g) {
    g.drawString(msg, X, Y);
}
}

```

Sample output is shown here:



The procedures shown in the preceding keyboard and mouse event examples can be generalized to any type of event handling, including those events generated by controls. In later chapters, you will see many examples that handle other types of events, but they will all follow the same basic structure as the programs just described.

Adapter Classes

Java provides a special feature, called an *adapter class*, that can simplify the creation of event handlers in certain situations. An adapter class provides an empty implementation of all methods in an event listener interface. Adapter classes are useful when you want to receive and process only some of the events that are handled by a particular event listener interface. You can define a new class to act as an event listener by extending one of the adapter classes and implementing only those events in which you are interested.

For example, the **MouseMotionAdapter** class has two methods, **mouseDragged()** and **mouseMoved()**, which are the methods defined by the **MouseMotionListener** interface. If you were interested in only mouse drag events, then you could simply extend **MouseMotionAdapter** and override **mouseDragged()**. The empty implementation of **mouseMoved()** would handle the mouse motion events for you.

Table 24-4 lists several commonly used adapter classes in **java.awt.event** and notes the interface that each implements.

Adapter Class	Listener Interface
ComponentAdapter	ComponentListener
ContainerAdapter	ContainerListener
FocusAdapter	FocusListener
KeyAdapter	KeyListener
MouseAdapter	MouseListener and (as of JDK 6) MouseMotionListener and MouseWheelListener
MouseMotionAdapter	MouseMotionListener
WindowAdapter	WindowListener, WindowFocusListener, and WindowStateListener

Table 24-4 Commonly Used Listener Interfaces Implemented by Adapter Classes

The following example demonstrates an adapter. It displays a message in the status bar of an applet viewer or browser when the mouse is clicked or dragged. However, all other mouse events are silently ignored. The program has three classes. **AdapterDemo** extends **Applet**. Its **init()** method creates an instance of **MyMouseAdapter** and registers that object to receive notifications of mouse events. It also creates an instance of **MyMouseMotionAdapter** and registers that object to receive notifications of mouse motion events. Both of the constructors take a reference to the applet as an argument.

MyMouseAdapter extends **MouseAdapter** and overrides the **mouseClicked()** method. The other mouse events are silently ignored by code inherited from the **MouseAdapter** class. **MyMouseMotionAdapter** extends **MouseMotionAdapter** and overrides the **mouseDragged()** method. The other mouse motion event is silently ignored by code inherited from the **MouseMotionAdapter** class. (**MouseAdaptor** also provides an empty implementation for **MouseMotionListener**. However, for the sake of illustration, this example handles each separately.)

Note that both of the event listener classes save a reference to the applet. This information is provided as an argument to their constructors and is used later to invoke the **showStatus()** method.

```
// Demonstrate an adapter.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="AdapterDemo" width=300 height=100>
</applet>
*/
public class AdapterDemo extends Applet {
    public void init() {
        addMouseListener(new MyMouseAdapter(this));
        addMouseMotionListener(new MyMouseMotionAdapter(this));
    }
}
```

```

class MyMouseAdapter extends MouseAdapter {
    AdapterDemo adapterDemo;
    public MyMouseAdapter(AdapterDemo adapterDemo) {
        this.adapterDemo = adapterDemo;
    }
    // Handle mouse clicked.
    public void mouseClicked(MouseEvent me) {
        adapterDemo.showStatus("Mouse clicked");
    }
}

class MyMouseMotionAdapter extends MouseMotionAdapter {
    AdapterDemo adapterDemo;
    public MyMouseMotionAdapter(AdapterDemo adapterDemo) {
        this.adapterDemo = adapterDemo;
    }
    // Handle mouse dragged.
    public void mouseDragged(MouseEvent me) {
        adapterDemo.showStatus("Mouse dragged");
    }
}

```

As you can see by looking at the program, not having to implement all of the methods defined by the **MouseMotionListener** and **MouseListener** interfaces saves you a considerable amount of effort and prevents your code from becoming cluttered with empty methods. As an exercise, you might want to try rewriting one of the keyboard input examples shown earlier so that it uses a **KeyAdapter**.

Inner Classes

In Chapter 7, the basics of inner classes were explained. Here, you will see why they are important. Recall that an *inner class* is a class defined within another class, or even within an expression. This section illustrates how inner classes can be used to simplify the code when using event adapter classes.

To understand the benefit provided by inner classes, consider the applet shown in the following listing. It *does not* use an inner class. Its goal is to display the string "Mouse Pressed" in the status bar of the applet viewer or browser when the mouse is pressed. There are two top-level classes in this program. **MousePressedDemo** extends **Applet**, and **MyMouseAdapter** extends **MouseAdapter**. The **init()** method of **MousePressedDemo** instantiates **MyMouseAdapter** and provides this object as an argument to the **addMouseListener()** method.

Notice that a reference to the applet is supplied as an argument to the **MyMouseAdapter** constructor. This reference is stored in an instance variable for later use by the **mousePressed()** method. When the mouse is pressed, it invokes the **showStatus()** method of the applet

through the stored applet reference. In other words, **showStatus()** is invoked relative to the applet reference stored by **MyMouseAdapter**.

```
// This applet does NOT use an inner class.
import java.applet.*;
import java.awt.event.*;
/*
<applet code="MousePressedDemo" width=200 height=100>
</applet>
*/
public class MousePressedDemo extends Applet {
    public void init() {
        addMouseListener(new MyMouseAdapter(this));
    }
}

class MyMouseAdapter extends MouseAdapter {
    MousePressedDemo mousePressedDemo;
    public MyMouseAdapter(MousePressedDemo mousePressedDemo) {
        this.mousePressedDemo = mousePressedDemo;
    }
    public void mousePressed(MouseEvent me) {
        mousePressedDemo.showStatus("Mouse Pressed.");
    }
}
```

The following listing shows how the preceding program can be improved by using an inner class. Here, **InnerClassDemo** is a top-level class that extends **Applet**. **MyMouseAdapter** is an inner class that extends **MouseAdapter**. Because **MyMouseAdapter** is defined within the scope of **InnerClassDemo**, it has access to all of the variables and methods within the scope of that class. Therefore, the **mousePressed()** method can call the **showStatus()** method directly. It no longer needs to do this via a stored reference to the applet. Thus, it is no longer necessary to pass **MyMouseAdapter()** a reference to the invoking object.

```
// Inner class demo.
import java.applet.*;
import java.awt.event.*;
/*
<applet code="InnerClassDemo" width=200 height=100>
</applet>
*/

public class InnerClassDemo extends Applet {
    public void init() {
        addMouseListener(new MyMouseAdapter());
    }
    class MyMouseAdapter extends MouseAdapter {
        public void mousePressed(MouseEvent me) {
            showStatus("Mouse Pressed");
        }
    }
}
```

Anonymous Inner Classes

An *anonymous* inner class is one that is not assigned a name. This section illustrates how an anonymous inner class can facilitate the writing of event handlers. Consider the applet shown in the following listing. As before, its goal is to display the string "Mouse Pressed" in the status bar of the applet viewer or browser when the mouse is pressed.

```
// Anonymous inner class demo.  
import java.applet.*;  
import java.awt.event.*;  
/*  
 <applet code="AnonymousInnerClassDemo" width=200 height=100>  
 </applet>  
 */  
  
public class AnonymousInnerClassDemo extends Applet {  
    public void init() {  
        addMouseListener(new MouseAdapter() {  
            public void mousePressed(MouseEvent me) {  
                showStatus("Mouse Pressed");  
            }  
        });  
    }  
}
```

There is one top-level class in this program: **AnonymousInnerClassDemo**. The **init()** method calls the **addMouseListener()** method. Its argument is an expression that defines and instantiates an anonymous inner class. Let's analyze this expression carefully.

The syntax **new MouseAdapter(){...}** indicates to the compiler that the code between the braces defines an anonymous inner class. Furthermore, that class extends **MouseAdapter**. This new class is not named, but it is automatically instantiated when this expression is executed.

Because this anonymous inner class is defined within the scope of **AnonymousInnerClassDemo**, it has access to all of the variables and methods within the scope of that class. Therefore, it can call the **showStatus()** method directly.

As just illustrated, both named and anonymous inner classes solve some annoying problems in a simple yet effective way. They also allow you to create more efficient code.

This page has been intentionally left blank

CHAPTER

25

Introducing the AWT: Working with Windows, Graphics, and Text

The Abstract Window Toolkit (AWT) was Java's first GUI framework, and it has been part of Java since version 1.0. It contains numerous classes and methods that allow you to create windows and simple controls. The AWT was introduced in Chapter 23, where it was used in several short, example applets. This chapter begins a more detailed examination. Here, you will learn how to create and manage windows, manage fonts, output text, and utilize graphics. Chapter 26 describes various AWT controls, such as scroll bars and push buttons. It also explains further aspects of Java's event handling mechanism. Chapter 27 introduces the AWT's imaging subsystem.

It is important to state at the outset that you will seldom create GUIs based solely on the AWT because more powerful GUI frameworks (Swing and JavaFX) have been developed for Java. Despite this fact, the AWT remains an important part of Java. To understand why, consider the following.

At the time of this writing, the framework that is most widely used is Swing. Because Swing provides a richer, more flexible GUI framework than does the AWT, it is easy to jump to the conclusion that the AWT is no longer relevant—that it has been fully superseded by Swing. This assumption is, however, false. Instead, an understanding of the AWT is still important because the AWT underpins Swing, with many AWT classes being used either directly or indirectly by Swing. As a result, a solid knowledge of the AWT is still required to use Swing effectively.

Java's newest GUI framework is JavaFX. It is anticipated that, at some point in the future, JavaFX will replace Swing as Java's most popular GUI. Even when this occurs, however, much legacy code that relies on Swing (and thus, the AWT) will still need to be maintained for some time to come. Finally, for some types of small programs (especially small applets) that make only minimal use of a GUI, using the AWT may still be appropriate. Therefore, even though the AWT constitutes Java's oldest GUI framework, a basic working knowledge of its fundamentals is still important today.

Although a common use of the AWT is in applets, it is also used to create stand-alone windows that run in a GUI environment, such as Windows. For the sake of convenience, most of the examples in this chapter are contained in applets. The easiest way to run them

is with the applet viewer. A few examples demonstrate the creation of stand-alone, windowed programs, which can be executed directly.

One last point before beginning: The AWT is quite large and a full description would easily fill an entire book. Therefore, it is not possible to describe in detail every AWT class, method, or instance variable. However, this and the following chapters explain the basic techniques needed to use the AWT. From there, you will be able to explore other parts of the AWT on your own. You will also be ready to move on to Swing.

NOTE If you have not yet read Chapter 24, please do so now. It provides an overview of event handling, which is used by many of the examples in this chapter.

AWT Classes

The AWT classes are contained in the **java.awt** package. It is one of Java's largest packages. Fortunately, because it is logically organized in a top-down, hierarchical fashion, it is easier to understand and use than you might at first believe. Table 25-1 lists some of the many AWT classes.

Class	Description
AWTEvent	Encapsulates AWT events.
AWTEventMulticaster	Dispatches events to multiple listeners.
BorderLayout	The border layout manager. Border layouts use five components: North, South, East, West, and Center.
Button	Creates a push button control.
Canvas	A blank, semantics-free window.
CardLayout	The card layout manager. Card layouts emulate index cards. Only the one on top is showing.
Checkbox	Creates a check box control.
CheckboxGroup	Creates a group of check box controls.
CheckboxMenuItem	Creates an on/off menu item.
Choice	Creates a pop-up list.
Color	Manages colors in a portable, platform-independent fashion.
Component	An abstract superclass for various AWT components.
Container	A subclass of Component that can hold other components.
Cursor	Encapsulates a bitmapped cursor.
Dialog	Creates a dialog window.
Dimension	Specifies the dimensions of an object. The width is stored in width , and the height is stored in height .
EventQueue	Queues events.
FileDialog	Creates a window from which a file can be selected.

Table 25-1 A Sampling of AWT Classes

Class	Description
FlowLayout	The flow layout manager. Flow layout positions components left to right, top to bottom.
Font	Encapsulates a type font.
FontMetrics	Encapsulates various information related to a font. This information helps you display text in a window.
Frame	Creates a standard window that has a title bar, resize corners, and a menu bar.
Graphics	Encapsulates the graphics context. This context is used by the various output methods to display output in a window.
GraphicsDevice	Describes a graphics device such as a screen or printer.
GraphicsEnvironment	Describes the collection of available Font and GraphicsDevice objects.
GridBagConstraints	Defines various constraints relating to the GridBagLayout class.
GridBagLayout	The grid bag layout manager. Grid bag layout displays components subject to the constraints specified by GridBagConstraints .
GridLayout	The grid layout manager. Grid layout displays components in a two-dimensional grid.
Image	Encapsulates graphical images.
Insets	Encapsulates the borders of a container.
Label	Creates a label that displays a string.
List	Creates a list from which the user can choose. Similar to the standard Windows list box.
MediaTracker	Manages media objects.
Menu	Creates a pull-down menu.
MenuBar	Creates a menu bar.
MenuComponent	An abstract class implemented by various menu classes.
MenuItem	Creates a menu item.
MenuShortcut	Encapsulates a keyboard shortcut for a menu item.
Panel	The simplest concrete subclass of Container .
Point	Encapsulates a Cartesian coordinate pair, stored in x and y .
Polygon	Encapsulates a polygon.
PopupMenu	Encapsulates a pop-up menu.
PrintJob	An abstract class that represents a print job.
Rectangle	Encapsulates a rectangle.
Robot	Supports automated testing of AWT-based applications.
Scrollbar	Creates a scroll bar control.
ScrollPane	A container that provides horizontal and/or vertical scroll bars for another component.

Table 25-1 A Sampling of AWT Classes (continued)

Class	Description
SystemColor	Contains the colors of GUI widgets such as windows, scroll bars, text, and others.
TextArea	Creates a multiline edit control.
TextComponent	A superclass for TextArea and TextField .
TextField	Creates a single-line edit control.
Toolkit	Abstract class implemented by the AWT.
Window	Creates a window with no frame, no menu bar, and no title.

Table 25-1 A Sampling of AWT Classes (*continued*)

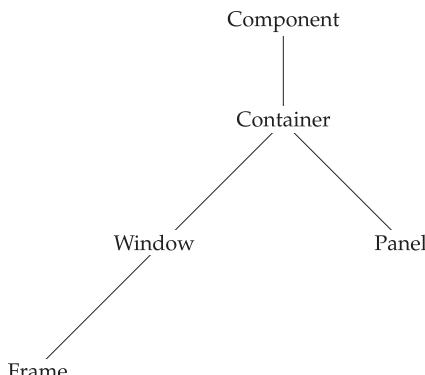
Although the basic structure of the AWT has been the same since Java 1.0, some of the original methods were deprecated and replaced by new ones. For backward-compatibility, Java still supports all the original 1.0 methods. However, because these methods are not for use with new code, this book does not describe them.

Window Fundamentals

The AWT defines windows according to a class hierarchy that adds functionality and specificity with each level. The two most common windows are those derived from **Panel**, which is used by applets, and those derived from **Frame**, which creates a standard application window. Much of the functionality of these windows is derived from their parent classes. Thus, a description of the class hierarchies relating to these two classes is fundamental to their understanding. Figure 25-1 shows the class hierarchy for **Panel** and **Frame**. Let's look at each of these classes now.

Component

At the top of the AWT hierarchy is the **Component** class. **Component** is an abstract class that encapsulates all of the attributes of a visual component. Except for menus, all user interface elements that are displayed on the screen and that interact with the user are

**Figure 25-1** The class hierarchy for **Panel** and **Frame**

subclasses of **Component**. It defines over a hundred public methods that are responsible for managing events, such as mouse and keyboard input, positioning and sizing the window, and repainting. (You already used many of these methods when you created applets in Chapters 23 and 24.) A **Component** object is responsible for remembering the current foreground and background colors and the currently selected text font.

Container

The **Container** class is a subclass of **Component**. It has additional methods that allow other **Component** objects to be nested within it. Other **Container** objects can be stored inside of a **Container** (since they are themselves instances of **Component**). This makes for a multileveled containment system. A container is responsible for laying out (that is, positioning) any components that it contains. It does this through the use of various layout managers, which you will learn about in Chapter 26.

Panel

The **Panel** class is a concrete subclass of **Container**. A **Panel** may be thought of as a recursively nestable, concrete screen component. **Panel** is the superclass for **Applet**. When screen output is directed to an applet, it is drawn on the surface of a **Panel** object. In essence, a **Panel** is a window that does not contain a title bar, menu bar, or border. This is why you don't see these items when an applet is run inside a browser. When you run an applet using an applet viewer, the applet viewer provides the title and border.

Other components can be added to a **Panel** object by its **add()** method (inherited from **Container**). Once these components have been added, you can position and resize them manually using the **setLocation()**, **setSize()**, **setPreferredSize()**, or **setBounds()** methods defined by **Component**.

Window

The **Window** class creates a top-level window. A *top-level window* is not contained within any other object; it sits directly on the desktop. Generally, you won't create **Window** objects directly. Instead, you will use a subclass of **Window** called **Frame**, described next.

Frame

Frame encapsulates what is commonly thought of as a "window." It is a subclass of **Window** and has a title bar, menu bar, borders, and resizing corners. The precise look of a **Frame** will differ among environments. A number of environments are reflected in the screen captures shown throughout this book.

Canvas

Although it is not part of the hierarchy for applet or frame windows, there is one other type of window that you will find valuable: **Canvas**. Derived from **Component**, **Canvas** encapsulates a blank window upon which you can draw. You will see an example of **Canvas** later in this book.

Working with Frame Windows

In addition to the applet, the type of AWT-based window you will most often create is derived from **Frame**. You will use it to create child windows within applets, and top-level or child windows for stand-alone applications. As mentioned, it creates a standard-style window.

Here are two of **Frame**'s constructors:

```
Frame( ) throws HeadlessException
Frame(String title) throws HeadlessException
```

The first form creates a standard window that does not contain a title. The second form creates a window with the title specified by *title*. Notice that you cannot specify the dimensions of the window. Instead, you must set the size of the window after it has been created. A **HeadlessException** is thrown if an attempt is made to create a **Frame** instance in an environment that does not support user interaction.

There are several key methods you will use when working with **Frame** windows. They are examined here.

Setting the Window's Dimensions

The **setSize()** method is used to set the dimensions of the window. Its signature is shown here:

```
void setSize(int newWidth, int newHeight)
void setSize(Dimension newSize)
```

The new size of the window is specified by *newWidth* and *newHeight*, or by the **width** and **height** fields of the **Dimension** object passed in *newSize*. The dimensions are specified in terms of pixels.

The **getSize()** method is used to obtain the current size of a window. One of its forms is shown here:

```
Dimension getSize( )
```

This method returns the current size of the window contained within the **width** and **height** fields of a **Dimension** object.

Hiding and Showing a Window

After a frame window has been created, it will not be visible until you call **setVisible()**. Its signature is shown here:

```
void setVisible(boolean visibleFlag)
```

The component is visible if the argument to this method is **true**. Otherwise, it is hidden.

Setting a Window's Title

You can change the title in a frame window using **setTitle()**, which has this general form:

```
void setTitle(String newTitle)
```

Here, *newTitle* is the new title for the window.

Closing a Frame Window

When using a frame window, your program must remove that window from the screen when it is closed, by calling `setVisible(false)`. To intercept a window-close event, you must implement the `windowClosing()` method of the **WindowListener** interface. Inside `windowClosing()`, you must remove the window from the screen. The example in the next section illustrates this technique.

Creating a Frame Window in an AWT-Based Applet

While it is possible to simply create a window by creating an instance of **Frame**, you will seldom do so, because you will not be able to do much with it. For example, you will not be able to receive or process events that occur within it or easily output information to it. Most of the time, you will create a subclass of **Frame**. Doing so lets you override **Frame**'s methods and provide event handling.

Creating a new frame window from within an AWT-based applet is actually quite easy. First, create a subclass of **Frame**. Next, override any of the standard applet methods, such as `init()`, `start()`, and `stop()`, to show or hide the frame as needed. Finally, implement the `windowClosing()` method of the **WindowListener** interface, calling `setVisible(false)` when the window is closed.

Once you have defined a **Frame** subclass, you can create an object of that class. This causes a frame window to come into existence, but it will not be initially visible. You make it visible by calling `setVisible()`. When created, the window is given a default height and width. You can set the size of the window explicitly by calling the `setSize()` method.

The following applet creates a subclass of **Frame** called **SampleFrame**. A window of this subclass is instantiated within the `init()` method of **AppletFrame**. Notice that **SampleFrame** calls **Frame**'s constructor. This causes a standard frame window to be created with the title passed in `title`. This example overrides the applet's `start()` and `stop()` methods so that they show and hide the child window, respectively. This causes the window to be removed automatically when you terminate the applet, when you close the window, or, if using a browser, when you move to another page. It also causes the child window to be shown when the browser returns to the applet.

```
// Create a child frame window from within an applet.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="AppletFrame" width=300 height=50>
</applet>
*/
// Create a subclass of Frame.
class SampleFrame extends Frame {
    SampleFrame(String title) {
        super(title);

        // create an object to handle window events
        MyWindowAdapter adapter = new MyWindowAdapter(this);
```

```
// register it to receive those events
    addWindowListener(adapter);
}
public void paint(Graphics g) {
    g.drawString("This is in frame window", 10, 40);
}
}

class MyWindowAdapter extends WindowAdapter {
    SampleFrame sampleFrame;

    public MyWindowAdapter(SampleFrame sampleFrame) {
        this.sampleFrame = sampleFrame;
    }

    public void windowClosing(WindowEvent we) {
        sampleFrame.setVisible(false);
    }
}

// Create frame window.
public class AppletFrame extends Applet {
    Frame f;
    public void init() {
        f = new SampleFrame("A Frame Window");

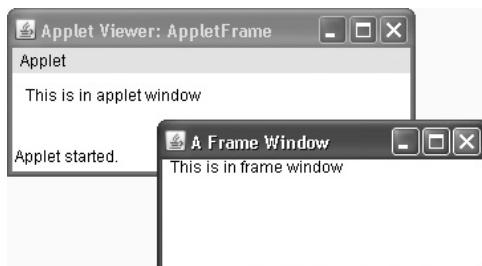
        f.setSize(250, 250);
        f.setVisible(true);
    }

    public void start() {
        f.setVisible(true);
    }

    public void stop() {
        f.setVisible(false);
    }

    public void paint(Graphics g) {
        g.drawString("This is in applet window", 10, 20);
    }
}
```

Sample output from this program is shown here:



Handling Events in a Frame Window

Since **Frame** is a subclass of **Component**, it inherits all the capabilities defined by **Component**. This means that you can use and manage a frame window just like you manage an applet's main window, as described earlier in this book. For example, you can override **paint()** to display output, call **repaint()** when you need to restore the window, and add event handlers. Whenever an event occurs in a window, the event handlers defined by that window will be called. Each window handles its own events. For example, the following program creates a window that responds to mouse events. The main applet window also responds to mouse events. When you experiment with this program, you will see that mouse events are sent to the window in which the event occurs.

```
// Handle mouse events in both child and applet windows.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="WindowEvents" width=300 height=50>
</applet>
*/
// Create a subclass of Frame.
class SampleFrame extends Frame
    implements MouseListener, MouseMotionListener {

    String msg = "";
    int mouseX=10, mouseY=40;
    int movX=0, movY=0;

    SampleFrame(String title) {
        super(title);
        // register this object to receive its own mouse events
        addMouseListener(this);
        addMouseMotionListener(this);
        // create an object to handle window events
        MyWindowAdapter adapter = new MyWindowAdapter(this);
        // register it to receive those events
        addWindowListener(adapter);
    }

    // Handle mouse clicked.
    public void mouseClicked(MouseEvent me) {
    }

    // Handle mouse entered.
    public void mouseEntered(MouseEvent evtObj) {
        // save coordinates
        mouseX = 10;
        mouseY = 54;
        msg = "Mouse just entered child.";
        repaint();
    }
}
```

```
// Handle mouse exited.
public void mouseExited(MouseEvent evtObj) {
    // save coordinates
    mouseX = 10;
    mouseY = 54;
    msg = "Mouse just left child window.";
    repaint();
}

// Handle mouse pressed.
public void mousePressed(MouseEvent me) {
    // save coordinates
    mouseX = me.getX();
    mouseY = me.getY();
    msg = "Down";
    repaint();
}

// Handle mouse released.
public void mouseReleased(MouseEvent me) {
    // save coordinates
    mouseX = me.getX();
    mouseY = me.getY();
    msg = "Up";
    repaint();
}

// Handle mouse dragged.
public void mouseDragged(MouseEvent me) {
    // save coordinates
    mouseX = me.getX();
    mouseY = me.getY();
    movX = me.getX();
    movY = me.getY();
    msg = "*";
    repaint();
}

// Handle mouse moved.
public void mouseMoved(MouseEvent me) {
    // save coordinates
    movX = me.getX();
    movY = me.getY();
    repaint(0, 0, 100, 60);
}

public void paint(Graphics g) {
    g.drawString(msg, mouseX, mouseY);
    g.drawString("Mouse at " + movX + ", " + movY, 10, 40);
}
}

class MyWindowAdapter extends WindowAdapter {
    SampleFrame sampleFrame;
```

```
public MyWindowAdapter(SampleFrame sampleFrame) {
    this.sampleFrame = sampleFrame;
}

public void windowClosing(WindowEvent we) {
    sampleFrame.setVisible(false);
}
}

// Applet window.
public class WindowEvents extends Applet
    implements MouseListener, MouseMotionListener {

    SampleFrame f;
    String msg = "";
    int mouseX=0, mouseY=10;
    int movX=0, movY=0;

    // Create a frame window.
    public void init() {
        f = new SampleFrame("Handle Mouse Events");
        f.setSize(300, 200);
        f.setVisible(true);

        // register this object to receive its own mouse events
        addMouseListener(this);
        addMouseMotionListener(this);
    }

    // Remove frame window when stopping applet.
    public void stop() {
        f.setVisible(false);
    }

    // Show frame window when starting applet.
    public void start() {
        f.setVisible(true);
    }

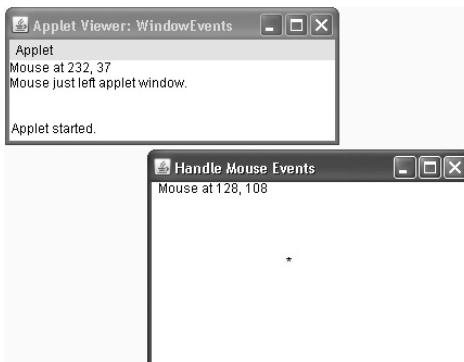
    // Handle mouse clicked.
    public void mouseClicked(MouseEvent me) {

    }

    // Handle mouse entered.
    public void mouseEntered(MouseEvent me) {
        // save coordinates
        mouseX = 0;
        mouseY = 24;
        msg = "Mouse just entered applet window.";
        repaint();
    }
}
```

```
// Handle mouse exited.  
public void mouseExited(MouseEvent me) {  
    // save coordinates  
    mouseX = 0;  
    mouseY = 24;  
    msg = "Mouse just left applet window.";  
    repaint();  
}  
  
// Handle button pressed.  
public void mousePressed(MouseEvent me) {  
    // save coordinates  
    mouseX = me.getX();  
    mouseY = me.getY();  
    msg = "Down";  
    repaint();  
}  
  
// Handle button released.  
public void mouseReleased(MouseEvent me) {  
    // save coordinates  
    mouseX = me.getX();  
    mouseY = me.getY();  
    msg = "Up";  
    repaint();  
}  
  
// Handle mouse dragged.  
public void mouseDragged(MouseEvent me) {  
    // save coordinates  
    mouseX = me.getX();  
    mouseY = me.getY();  
    movX = me.getX();  
    movY = me.getY();  
    msg = "*";  
    repaint();  
}  
  
// Handle mouse moved.  
public void mouseMoved(MouseEvent me) {  
    // save coordinates  
    movX = me.getX();  
    movY = me.getY();  
    repaint(0, 0, 100, 20);  
}  
  
// Display msg in applet window.  
public void paint(Graphics g) {  
    g.drawString(msg, mouseX, mouseY);  
    g.drawString("Mouse at " + movX + ", " + movY, 0, 10);  
}  
}
```

Sample output from this program is shown here:



Creating a Windowed Program

Although creating applets is a common use for Java's AWT, it is also possible to create stand-alone AWT-based applications. To do this, simply create an instance of the window or windows you need inside `main()`. For example, the following program creates a frame window that responds to mouse clicks and keystrokes:

```
// Create an AWT-based application.  
import java.awt.*;  
import java.awt.event.*;  
import java.applet.*;  
  
// Create a frame window.  
public class AppWindow extends Frame {  
    String keymsg = "This is a test.";  
    String mousemsg = "";  
    int mouseX=30, mouseY=30;  
  
    public AppWindow() {  
        addKeyListener(new MyKeyAdapter(this));  
        addMouseListener(new MyMouseAdapter(this));  
        addWindowListener(new MyWindowAdapter());  
    }  
  
    public void paint(Graphics g) {  
        g.drawString(keymsg, 10, 40);  
        g.drawString(mousemsg, mouseX, mouseY);  
    }  
  
    // Create the window.  
    public static void main(String args[]) {  
        AppWindow appwin = new AppWindow();  
  
        appwin.setSize(new Dimension(300, 200));  
        appwin.setTitle("An AWT-Based Application");  
        appwin.setVisible(true);  
    }  
}
```

```
class MyKeyAdapter extends KeyAdapter {
    AppWindow appWindow;

    public MyKeyAdapter(AppWindow appWindow) {
        this.appWindow = appWindow;
    }

    public void keyTyped(KeyEvent ke) {
        appWindow.keymsg += ke.getKeyChar();
        appWindow.repaint();
    }
}

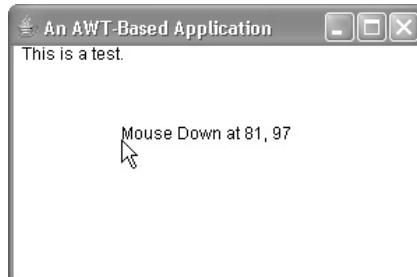
class MyMouseAdapter extends MouseAdapter {
    AppWindow appWindow;

    public MyMouseAdapter(AppWindow appWindow) {
        this.appWindow = appWindow;
    }

    public void mousePressed(MouseEvent me) {
        appWindow.mouseX = me.getX();
        appWindow.mouseY = me.getY();
        appWindow.mousemsg = "Mouse Down at " + appWindow.mouseX +
            ", " + appWindow.mouseY;
        appWindow.repaint();
    }
}

class MyWindowAdapter extends WindowAdapter {
    public void windowClosing(WindowEvent we) {
        System.exit(0);
    }
}
```

Sample output from this program is shown here:



Once created, a frame window takes on a life of its own. Notice that `main()` ends with the call to `appwin.setVisible(true)`. However, the program keeps running until you close the window. In essence, when creating a windowed application, you will use `main()` to launch its top-level window. After that, your program will function as a GUI-based application, not like the console-based programs used earlier.

Displaying Information Within a Window

In the most general sense, a window is a container for information. Although we have already output small amounts of text to a window in the preceding examples, we have not begun to take advantage of a window's ability to present high-quality text and graphics. Indeed, much of the power of the AWT comes from its support for these items. For this reason, the remainder of this chapter introduces the AWT's text-, graphics-, and font-handling capabilities. As you will see, they are both powerful and flexible.

Introducing Graphics

The AWT includes several methods that support graphics. All graphics are drawn relative to a window. This can be the main window of an applet, a child window of an applet, or a stand-alone application window. (These methods are also supported by Swing-based windows.) The origin of each window is at the top-left corner and is 0,0. Coordinates are specified in pixels. All output to a window takes place through a *graphics context*.

A graphics context is encapsulated by the **Graphics** class. Here are two ways in which a graphics context can be obtained:

- It is passed to a method, such as **paint()** or **update()**, as an argument.
- It is returned by the **getGraphics()** method of **Component**.

Among other things, the **Graphics** class defines a number of methods that draw various types of objects, such as lines, rectangles, and arcs. In several cases, objects can be drawn edge-only or filled. Objects are drawn and filled in the currently selected color, which is black by default. When a graphics object is drawn that exceeds the dimensions of the window, output is automatically clipped. A sampling of the drawing methods supported by **Graphics** is presented here.

NOTE With the release of version 1.2, the graphics capabilities of Java were expanded by the inclusion of several new classes. One of these is **Graphics2D**, which extends **Graphics**. **Graphics2D** supports several powerful enhancements to the basic capabilities provided by **Graphics**. To gain access to this extended functionality, you must cast the graphics context obtained from a method such as **paint()**, to **Graphics2D**. Although the basic graphics functions supported by **Graphics** are adequate for the purposes of this book, **Graphics2D** is a class that you will want to explore fully on your own if you will be programming sophisticated graphics applications.

Drawing Lines

Lines are drawn by means of the **drawLine()** method, shown here:

```
void drawLine(int startX, int startY, int endX, int endY)
```

drawLine() displays a line in the current drawing color that begins at *startX*, *startY* and ends at *endX*, *endY*.

Drawing Rectangles

The **drawRect()** and **fillRect()** methods display an outlined and filled rectangle, respectively. They are shown here:

```
void drawRect(int left, int top, int width, int height)
void fillRect(int left, int top, int width, int height)
```

The upper-left corner of the rectangle is at *left*, *top*. The dimensions of the rectangle are specified by *width* and *height*.

To draw a rounded rectangle, use **drawRoundRect()** or **fillRoundRect()**, both shown here:

```
void drawRoundRect(int left, int top, int width, int height,
int xDiam, int yDiam)
```

```
void fillRoundRect(int left, int top, int width, int height,
int xDiam, int yDiam)
```

A rounded rectangle has rounded corners. The upper-left corner of the rectangle is at *left*, *top*. The dimensions of the rectangle are specified by *width* and *height*. The diameter of the rounding arc along the X axis is specified by *xDiam*. The diameter of the rounding arc along the Y axis is specified by *yDiam*.

Drawing Ellipses and Circles

To draw an ellipse, use **drawOval()**. To fill an ellipse, use **fillOval()**. These methods are shown here:

```
void drawOval(int left, int top, int width, int height)
void fillOval(int left, int top, int width, int height)
```

The ellipse is drawn within a bounding rectangle whose upper-left corner is specified by *left*, *top* and whose width and height are specified by *width* and *height*. To draw a circle, specify a square as the bounding rectangle.

Drawing Arcs

Arcs can be drawn with **drawArc()** and **fillArc()**, shown here:

```
void drawArc(int left, int top, int width, int height, int startAngle,
int sweepAngle)
```

```
void fillArc(int left, int top, int width, int height, int startAngle,
int sweepAngle)
```

The arc is bounded by the rectangle whose upper-left corner is specified by *left*, *top* and whose width and height are specified by *width* and *height*. The arc is drawn from *startAngle* through the angular distance specified by *sweepAngle*. Angles are specified in degrees. Zero degrees is on the horizontal, at the three o'clock position. The arc is drawn counterclockwise if *sweepAngle* is positive, and clockwise if *sweepAngle* is negative. Therefore, to draw an arc from twelve o'clock to six o'clock, the start angle would be 90 and the sweep angle 180.

Drawing Polygons

It is possible to draw arbitrarily shaped figures using `drawPolygon()` and `fillPolygon()`, shown here:

```
void drawPolygon(int x[ ], int y[ ], int numPoints)
void fillPolygon(int x[ ], int y[ ], int numPoints)
```

The polygon's endpoints are specified by the coordinate pairs contained within the `x` and `y` arrays. The number of points defined by these arrays is specified by `numPoints`. There are alternative forms of these methods in which the polygon is specified by a `Polygon` object.

Demonstrating the Drawing Methods

The following program demonstrates the drawing methods just described.

```
// Draw graphics elements.
import java.awt.*;
import java.applet.*;
/*
<applet code="GraphicsDemo" width=350 height=700>
</applet>
*/
public class GraphicsDemo extends Applet {
    public void paint(Graphics g) {

        // Draw lines.
        g.drawLine(0, 0, 100, 90);
        g.drawLine(0, 90, 100, 10);
        g.drawLine(40, 25, 250, 80);

        // Draw rectangles.
        g.drawRect(10, 150, 60, 50);
        g.fillRect(100, 150, 60, 50);
        g.drawRoundRect(190, 150, 60, 50, 15, 15);
        g.fillRoundRect(280, 150, 60, 50, 30, 40);

        // Draw Ellipses and Circles
        g.drawOval(10, 250, 50, 50);
        g.fillOval(90, 250, 75, 50);
        g.drawOval(190, 260, 100, 40);

        // Draw Arcs
        g.drawArc(10, 350, 70, 70, 0, 180);
        g.fillArc(60, 350, 70, 70, 0, 75);

        // Draw a polygon
        int xpoints[] = {10, 200, 10, 200, 10};
        int ypoints[] = {450, 450, 650, 650, 450};
        int num = 5;

        g.drawPolygon(xpoints, ypoints, num);
    }
}
```

Sample output is shown in Figure 25-2.

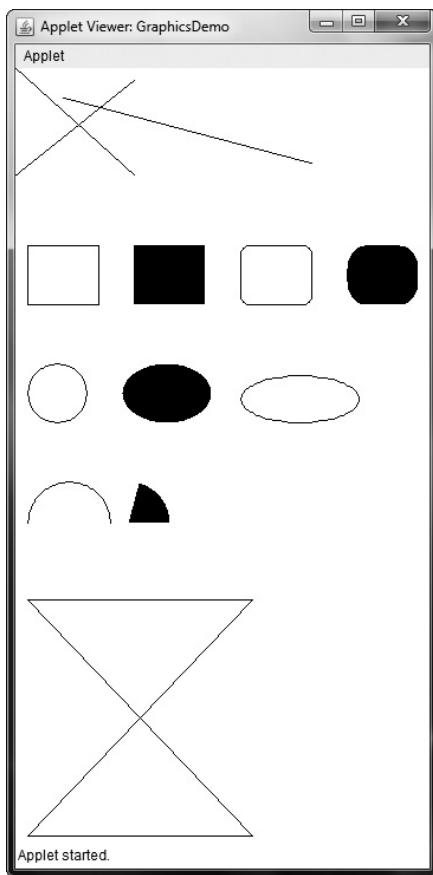


Figure 25-2 Sample output from the **GraphicsDemo** program

Sizing Graphics

Often, you will want to size a graphics object to fit the current size of the window in which it is drawn. To do so, first obtain the current dimensions of the window by calling `getSize()` on the window object. It returns the dimensions of the window encapsulated within a **Dimension** object. Once you have the current size of the window, you can scale your graphical output accordingly.

To demonstrate this technique, here is an applet that will start as a 200×200-pixel square and grow by 25 pixels in width and height with each mouse click until the applet gets larger than 500×500. At that point, the next click will return it to 200×200, and the process starts over.

Within the window, a rectangle is drawn around the inner border of the window; within that rectangle, an *X* is drawn so that it fills the window. This applet works in **appletviewer**, but it may not work in a browser window.

```
// Resizing output to fit the current size of a window.  
import java.applet.*;
```

```

import java.awt.*;
import java.awt.event.*;
/*
<applet code="ResizeMe" width=200 height=200>
</applet>
*/
public class ResizeMe extends Applet {
    final int inc = 25;
    int max = 500;
    int min = 200;
    Dimension d;
    public ResizeMe() {
        addMouseListener(new MouseAdapter() {
            public void mouseReleased(MouseEvent me) {
                int w = (d.width + inc) > max?min :(d.width + inc);
                int h = (d.height + inc) > max?min :(d.height + inc);
                setSize(new Dimension(w, h));
            }
        });
    }
    public void paint(Graphics g) {
        d = getSize();
        g.drawLine(0, 0, d.width-1, d.height-1);
        g.drawLine(0, d.height-1, d.width-1, 0);
        g.drawRect(0, 0, d.width-1, d.height-1);
    }
}

```

Working with Color

Java supports color in a portable, device-independent fashion. The AWT color system allows you to specify any color you want. It then finds the best match for that color, given the limits of the display hardware currently executing your program or applet. Thus, your code does not need to be concerned with the differences in the way color is supported by various hardware devices. Color is encapsulated by the **Color** class.

As you saw in Chapter 23, **Color** defines several constants (for example, **Color.black**) to specify a number of common colors. You can also create your own colors, using one of the color constructors. Three commonly used forms are shown here:

```

Color(int red, int green, int blue)
Color(int rgbValue)
Color(float red, float green, float blue)

```

The first constructor takes three integers that specify the color as a mix of red, green, and blue. These values must be between 0 and 255, as in this example:

```
new Color(255, 100, 100); // light red
```

The second color constructor takes a single integer that contains the mix of red, green, and blue packed into an integer. The integer is organized with red in bits 16 to 23, green in bits 8 to 15, and blue in bits 0 to 7. Here is an example of this constructor:

```
int newRed = (0xff000000 | (0xc0 << 16) | (0x00 << 8) | 0x00);
Color darkRed = new Color(newRed);
```

The final constructor, **Color(float, float, float)**, takes three **float** values (between 0.0 and 1.0) that specify the relative mix of red, green, and blue.

Once you have created a color, you can use it to set the foreground and/or background color by using the **setForeground()** and **setBackground()** methods described in Chapter 23. You can also select it as the current drawing color.

Color Methods

The **Color** class defines several methods that help manipulate colors. Several are examined here.

Using Hue, Saturation, and Brightness

The *hue-saturation-brightness (HSB)* color model is an alternative to red-green-blue (RGB) for specifying particular colors. Figuratively, *hue* is a wheel of color. The hue can be specified with a number between 0.0 and 1.0, which is used to obtain an angle into the color wheel. (The principal colors are approximately red, orange, yellow, green, blue, indigo, and violet.) *Saturation* is another scale ranging from 0.0 to 1.0, representing light pastels to intense hues. *Brightness* values also range from 0.0 to 1.0, where 1 is bright white and 0 is black. **Color** supplies two methods that let you convert between RGB and HSB. They are shown here:

```
static int HSBtoRGB(float hue, float saturation, float brightness)
static float[ ] RGBtoHSB(int red, int green, int blue, float values[ ])
```

HSBtoRGB() returns a packed RGB value compatible with the **Color(int)** constructor.

RGBtoHSB() returns a **float** array of HSB values corresponding to RGB integers. If *values* is not **null**, then this array is given the HSB values and returned. Otherwise, a new array is created and the HSB values are returned in it. In either case, the array contains the hue at index 0, saturation at index 1, and brightness at index 2.

getRed(), getGreen(), getBlue()

You can obtain the red, green, and blue components of a color independently using **getRed()**, **getGreen()**, and **getBlue()**, shown here:

```
int getRed()
int getGreen()
int getBlue()
```

Each of these methods returns the RGB color component found in the invoking **Color** object in the lower 8 bits of an integer.

getRGB()

To obtain a packed, RGB representation of a color, use `getRGB()`, shown here:

```
int getRGB()
```

The return value is organized as described earlier.

Setting the Current Graphics Color

By default, graphics objects are drawn in the current foreground color. You can change this color by calling the **Graphics** method `setColor()`:

```
void setColor(Color newColor)
```

Here, `newColor` specifies the new drawing color.

You can obtain the current color by calling `getColor()`, shown here:

```
Color getColor()
```

A Color Demonstration Applet

The following applet constructs several colors and draws various objects using these colors:

```
// Demonstrate color.
import java.awt.*;
import java.applet.*;
/*
<applet code="ColorDemo" width=300 height=200>
</applet>
*/
public class ColorDemo extends Applet {
    // draw lines
    public void paint(Graphics g) {
        Color c1 = new Color(255, 100, 100);
        Color c2 = new Color(100, 255, 100);
        Color c3 = new Color(100, 100, 255);

        g.setColor(c1);
        g.drawLine(0, 0, 100, 100);
        g.drawLine(0, 100, 100, 0);

        g.setColor(c2);
        g.drawLine(40, 25, 250, 180);
        g.drawLine(75, 90, 400, 400);

        g.setColor(c3);
        g.drawLine(20, 150, 400, 40);
        g.drawLine(5, 290, 80, 19);

        g.setColor(Color.red);
        g.drawOval(10, 10, 50, 50);
        g.fillOval(70, 90, 140, 100);
```

```

        g.setColor(Color.blue);
        g.drawOval(190, 10, 90, 30);
        g.drawRect(10, 10, 60, 50);

        g.setColor(Color.cyan);
        g.fillRect(100, 10, 60, 50);
        g.drawRoundRect(190, 10, 60, 50, 15, 15);
    }
}

```

Setting the Paint Mode

The *paint mode* determines how objects are drawn in a window. By default, new output to a window overwrites any preexisting contents. However, it is possible to have new objects XORed onto the window by using **setXORMode()**, as follows:

```
void setXORMode(Color xorColor)
```

Here, *xorColor* specifies the color that will be XORed to the window when an object is drawn. The advantage of XOR mode is that the new object is always guaranteed to be visible no matter what color the object is drawn over.

To return to overwrite mode, call **setPaintMode()**, shown here:

```
void setPaintMode()
```

In general, you will want to use overwrite mode for normal output, and XOR mode for special purposes. For example, the following program displays cross hairs that track the mouse pointer. The cross hairs are XORed onto the window and are always visible, no matter what the underlying color is.

```

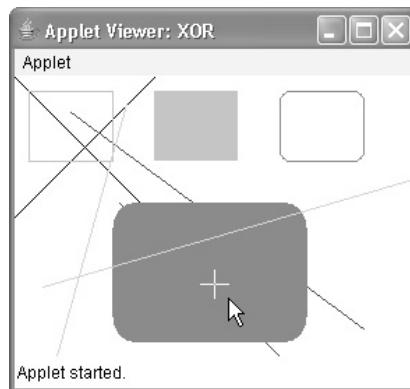
// Demonstrate XOR mode.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="XOR" width=400 height=200>
</applet>
*/
public class XOR extends Applet {
    int chsX=100, chsY=100;

    public XOR() {
        addMouseMotionListener(new MouseMotionAdapter() {
            public void mouseMoved(MouseEvent me) {
                int x = me.getX();
                int y = me.getY();
                chsX = x-10;
                chsY = y-10;
                repaint();
            }
        });
    }
}

```

```
public void paint(Graphics g) {  
    g.drawLine(0, 0, 100, 100);  
    g.drawLine(0, 100, 100, 0);  
    g.setColor(Color.blue);  
    g.drawLine(40, 25, 250, 180);  
    g.drawLine(75, 90, 400, 400);  
    g.setColor(Color.green);  
    g.drawRect(10, 10, 60, 50);  
    g.fillRect(100, 10, 60, 50);  
    g.setColor(Color.red);  
    g.drawRoundRect(190, 10, 60, 50, 15, 15);  
    g.fillRoundRect(70, 90, 140, 100, 30, 40);  
    g.setColor(Color.cyan);  
    g.drawLine(20, 150, 400, 40);  
    g.drawLine(5, 290, 80, 19);  
  
    // xor cross hairs  
    g.setXORMode(Color.black);  
    g.drawLine(chsX-10, chsY, chsX+10, chsY);  
    g.drawLine(chsX, chsY-10, chsX, chsY+10);  
    g.setPaintMode();  
}  
}
```

Sample output from this program is shown here:



Working with Fonts

The AWT supports multiple type fonts. Years ago, fonts emerged from the domain of traditional typesetting to become an important part of computer-generated documents and displays. The AWT provides flexibility by abstracting font-manipulation operations and allowing for dynamic selection of fonts.

Fonts have a family name, a logical font name, and a face name. The *family name* is the general name of the font, such as Courier. The *logical name* specifies a name, such as Monospaced, that is linked to an actual font at runtime. The *face name* specifies a specific font, such as Courier Italic.

Fonts are encapsulated by the **Font** class. Several of the methods defined by **Font** are listed in Table 25-2.

The **Font** class defines these protected variables:

Variable	Meaning
String name	Name of the font
float pointSize	Size of the font in points
int size	Size of the font in points
int style	Font style

Several static fields are also defined.

Method	Description
static Font decode(String <i>str</i>)	Returns a font given its name.
boolean equals(Object <i>FontObj</i>)	Returns true if the invoking object contains the same font as that specified by <i>FontObj</i> . Otherwise, it returns false .
String getFamily()	Returns the name of the font family to which the invoking font belongs.
static Font getFont(String <i>property</i>)	Returns the font associated with the system property specified by <i>property</i> . null is returned if <i>property</i> does not exist.
static Font getFont(String <i>property</i> , Font <i>defaultFont</i>)	Returns the font associated with the system property specified by <i>property</i> . The font specified by <i>defaultFont</i> is returned if <i>property</i> does not exist.
String getFontName()	Returns the face name of the invoking font.
String getName()	Returns the logical name of the invoking font.
int getSize()	Returns the size, in points, of the invoking font.
int getStyle()	Returns the style values of the invoking font.
int hashCode()	Returns the hash code associated with the invoking object.
boolean isBold()	Returns true if the font includes the BOLD style value. Otherwise, false is returned.
boolean isItalic()	Returns true if the font includes the ITALIC style value. Otherwise, false is returned.
boolean isPlain()	Returns true if the font includes the PLAIN style value. Otherwise, false is returned.
String toString()	Returns the string equivalent of the invoking font.

Table 25-2 A Sampling of Methods Defined by **Font**

Determining the Available Fonts

When working with fonts, often you need to know which fonts are available on your machine. To obtain this information, you can use the `getAvailableFontFamilyNames()` method defined by the **GraphicsEnvironment** class. It is shown here:

```
String[ ] getAvailableFontFamilyNames( )
```

This method returns an array of strings that contains the names of the available font families.

In addition, the `getAllFonts()` method is defined by the **GraphicsEnvironment** class. It is shown here:

```
Font[ ] getAllFonts( )
```

This method returns an array of **Font** objects for all of the available fonts.

Since these methods are members of **GraphicsEnvironment**, you need a **GraphicsEnvironment** reference to call them. You can obtain this reference by using the `getLocalGraphicsEnvironment()` static method, which is defined by **GraphicsEnvironment**. It is shown here:

```
static GraphicsEnvironment getLocalGraphicsEnvironment( )
```

Here is an applet that shows how to obtain the names of the available font families:

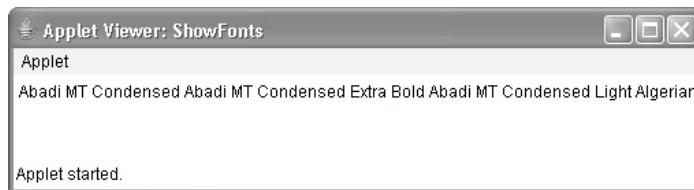
```
// Display Fonts
/*
<applet code="ShowFonts" width=550 height=60>
</applet>
*/
import java.applet.*;
import java.awt.*;

public class ShowFonts extends Applet {
    public void paint(Graphics g) {
        String msg = "";
        String FontList[] ;

        GraphicsEnvironment ge =
            GraphicsEnvironment.getLocalGraphicsEnvironment();
        FontList = ge.getAvailableFontFamilyNames();
        for(int i = 0; i < FontList.length; i++)
            msg += FontList[i] + " ";

        g.drawString(msg, 4, 16);
    }
}
```

Sample output from this program is shown next. However, when you run this program, you may see a different list of fonts than the one shown in this illustration.



Creating and Selecting a Font

To create a new font, construct a **Font** object that describes that font. One **Font** constructor has this general form:

```
Font(String fontName, int fontStyle, int pointSize)
```

Here, *fontName* specifies the name of the desired font. The name can be specified using either the logical or face name. All Java environments will support the following fonts: Dialog, DialogInput, SansSerif, Serif, and Monospaced. Dialog is the font used by your system's dialog boxes. Dialog is also the default if you don't explicitly set a font. You can also use any other fonts supported by your particular environment, but be careful—these other fonts may not be universally available.

The style of the font is specified by *fontStyle*. It may consist of one or more of these three constants: **Font.PLAIN**, **Font.BOLD**, and **Font.ITALIC**. To combine styles, OR them together. For example, **Font.BOLD | Font.ITALIC** specifies a bold, italics style.

The size, in points, of the font is specified by *pointSize*.

To use a font that you have created, you must select it using **setFont()**, which is defined by **Component**. It has this general form:

```
void setFont(Font fontObj)
```

Here, *fontObj* is the object that contains the desired font.

The following program outputs a sample of each standard font. Each time you click the mouse within its window, a new font is selected and its name is displayed.

```
// Show fonts.
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
/*
<applet code="SampleFonts" width=200 height=100>
</applet>
*/
public class SampleFonts extends Applet {
    int next = 0;
    Font f;
    String msg;
```

```
public void init() {
    f = new Font("Dialog", Font.PLAIN, 12);
    msg = "Dialog";
    setFont(f);
    addMouseListener(new MyMouseAdapter(this));
}

public void paint(Graphics g) {
    g.drawString(msg, 4, 20);
}
}

class MyMouseAdapter extends MouseAdapter {
    SampleFonts sampleFonts;

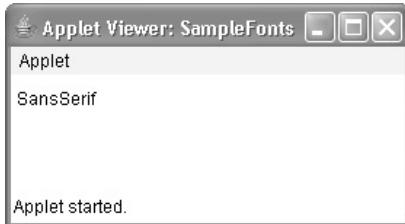
    public MyMouseAdapter(SampleFonts sampleFonts) {
        this.sampleFonts = sampleFonts;
    }

    public void mousePressed(MouseEvent me) {
        // Switch fonts with each mouse click.
        sampleFonts.next++;
        switch(sampleFonts.next) {
        case 0:
            sampleFonts.f = new Font("Dialog", Font.PLAIN, 12);
            sampleFonts.msg = "Dialog";
            break;
        case 1:
            sampleFonts.f = new Font("DialogInput", Font.PLAIN, 12);
            sampleFonts.msg = "DialogInput";
            break;
        case 2:
            sampleFonts.f = new Font("SansSerif", Font.PLAIN, 12);
            sampleFonts.msg = "SansSerif";
            break;
        case 3:
            sampleFonts.f = new Font("Serif", Font.PLAIN, 12);
            sampleFonts.msg = "Serif";
            break;
        case 4:
            sampleFonts.f = new Font("Monospaced", Font.PLAIN, 12);
            sampleFonts.msg = "Monospaced";
            break;
        }
    }

    if(sampleFonts.next == 4) sampleFonts.next = -1;

    sampleFonts.setFont(sampleFonts.f);
    sampleFonts.repaint();
}
}
```

Sample output from this program is shown here:



Obtaining Font Information

Suppose you want to obtain information about the currently selected font. To do this, you must first get the current font by calling `getFont()`. This method is defined by the **Graphics** class, as shown here:

```
Font getFont()
```

Once you have obtained the currently selected font, you can retrieve information about it using various methods defined by **Font**. For example, this applet displays the name, family, size, and style of the currently selected font:

```
// Display font info.
import java.applet.*;
import java.awt.*;
/*
<applet code="FontInfo" width=350 height=60>
</applet>
*/
public class FontInfo extends Applet {
    public void paint(Graphics g) {
        Font f = g.getFont();
        String fontName = f.getName();
        String fontFamily = f.getFamily();
        int fontSize = f.getSize();
        int fontStyle = f.getStyle();

        String msg = "Family: " + fontName;
        msg += ", Font: " + fontFamily;
        msg += ", Size: " + fontSize + ", Style: ";
        if((fontStyle & Font.BOLD) == Font.BOLD)
            msg += "Bold ";
        if((fontStyle & Font.ITALIC) == Font.ITALIC)
            msg += "Italic ";
        if((fontStyle & Font.PLAIN) == Font.PLAIN)
            msg += "Plain ";

        g.drawString(msg, 4, 16);
    }
}
```

Managing Text Output Using FontMetrics

As just explained, Java supports a number of fonts. For most fonts, characters are not all the same dimension—most fonts are proportional. Also, the height of each character, the length of *descenders* (the hanging parts of letters, such as *y*), and the amount of space between horizontal lines vary from font to font. Further, the point size of a font can be changed. That these (and other) attributes are variable would not be of too much consequence except that Java demands that you, the programmer, manually manage virtually all text output.

Given that the size of each font may differ and that fonts may be changed while your program is executing, there must be some way to determine the dimensions and various other attributes of the currently selected font. For example, to write one line of text after another implies that you have some way of knowing how tall the font is and how many pixels are needed between lines. To fill this need, the AWT includes the **FontMetrics** class, which encapsulates various information about a font. Let's begin by defining the common terminology used when describing fonts:

Height	The top-to-bottom size of a line of text
Baseline	The line that the bottoms of characters are aligned to (not counting descent)
Ascent	The distance from the baseline to the top of a character
Descent	The distance from the baseline to the bottom of a character
Leading	The distance between the bottom of one line of text and the top of the next

As you know, we have used the **drawString()** method in many of the previous examples. It paints a string in the current font and color, beginning at a specified location. However, this location is at the left edge of the baseline of the characters, not at the upper-left corner as is usual with other drawing methods. It is a common error to draw a string at the same coordinate that you would draw a box. For example, if you were to draw a rectangle at coordinate 0,0, you would see a full rectangle. If you were to draw the string “Typesetting” at 0,0, you would only see the tails (or descenders) of the *y*, *p*, and *g*. As you will see, by using font metrics, you can determine the proper placement of each string that you display.

FontMetrics defines several methods that help you manage text output. Several commonly used ones are listed in Table 25-3. These methods help you properly display text in a window. Let's look at some examples.

Displaying Multiple Lines of Text

Perhaps the most common use of **FontMetrics** is to determine the spacing between lines of text. The second most common use is to determine the length of a string that is being displayed. Here, you will see how to accomplish these tasks.

In general, to display multiple lines of text, your program must manually keep track of the current output position. Each time a newline is desired, the Y coordinate must be advanced to the beginning of the next line. Each time a string is displayed, the X coordinate must be set to the point at which the string ends. This allows the next string to be written so that it begins at the end of the preceding one.

Method	Description
int bytesWidth(byte <i>b</i> [], int <i>start</i> , int <i>numBytes</i>)	Returns the width of <i>numBytes</i> characters held in array <i>b</i> , beginning at <i>start</i> .
int charWidth(char <i>c</i> [], int <i>start</i> , int <i>numChars</i>)	Returns the width of <i>numChars</i> characters held in array <i>c</i> , beginning at <i>start</i> .
int charWidth(char <i>c</i>)	Returns the width of <i>c</i> .
int charWidth(int <i>c</i>)	Returns the width of <i>c</i> .
int getAscent()	Returns the ascent of the font.
int getDescent()	Returns the descent of the font.
Font getFont()	Returns the font.
int getHeight()	Returns the height of a line of text. This value can be used to output multiple lines of text in a window.
int getLeading()	Returns the space between lines of text.
int getMaxAdvance()	Returns the width of the widest character. -1 is returned if this value is not available.
int getMaxAscent()	Returns the maximum ascent.
int getMaxDescent()	Returns the maximum descent.
int[] getWidths()	Returns the widths of the first 256 characters.
int stringWidth(String <i>str</i>)	Returns the width of the string specified by <i>str</i> .
String toString()	Returns the string equivalent of the invoking object.

Table 25-3 A Sampling of Methods Defined by **FontMetrics**

To determine the spacing between lines, you can use the value returned by **getLeading()**. To determine the total height of the font, add the value returned by **getAscent()** to the value returned by **getDescent()**. You can then use these values to position each line of text you output. However, in many cases, you will not need to use these individual values. Often, all that you will need to know is the total height of a line, which is the sum of the leading space and the font's ascent and descent values. The easiest way to obtain this value is to call **getHeight()**. Simply increment the Y coordinate by this value each time you want to advance to the next line when outputting text.

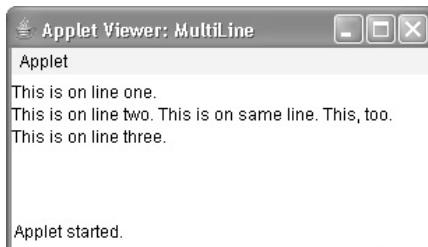
To start output at the end of previous output on the same line, you must know the length, in pixels, of each string that you display. To obtain this value, call **stringWidth()**. You can use this value to advance the X coordinate each time you display a line.

The following applet shows how to output multiple lines of text in a window. It also displays multiple sentences on the same line. Notice the variables **curX** and **curY**. They keep track of the current text output position.

```
// Demonstrate multiline output.
import java.applet.*;
import java.awt.*;
/*
<applet code="MultiLine" width=300 height=100>
</applet>
*/
```

```
public class MultiLine extends Applet {  
    int curX=0, curY=0; // current position  
  
    public void init() {  
        Font f = new Font("SansSerif", Font.PLAIN, 12);  
        setFont(f);  
    }  
  
    public void paint(Graphics g) {  
        FontMetrics fm = g.getFontMetrics();  
  
        nextLine("This is on line one.", g);  
        nextLine("This is on line two.", g);  
        sameLine(" This is on same line.", g);  
        sameLine(" This, too.", g);  
        nextLine("This is on line three.", g);  
        curX = curY = 0; // Reset coordinates for each repaint.  
    }  
  
    // Advance to next line.  
    void nextLine(String s, Graphics g) {  
        FontMetrics fm = g.getFontMetrics();  
  
        curY += fm.getHeight(); // advance to next line  
        curX = 0;  
        g.drawString(s, curX, curY);  
        curX = fm.stringWidth(s); // advance to end of line  
    }  
  
    // Display on same line.  
    void sameLine(String s, Graphics g) {  
        FontMetrics fm = g.getFontMetrics();  
  
        g.drawString(s, curX, curY);  
        curX += fm.stringWidth(s); // advance to end of line  
    }  
}
```

Sample output from this program is shown here:



Centering Text

Here is an example that centers text, left to right, top to bottom, in a window. It obtains the ascent, descent, and width of the string and computes the position at which it must be displayed to be centered.

```
// Center text.
import java.applet.*;
import java.awt.*;
/*
<applet code="CenterText" width=200 height=100>
</applet>
*/
public class CenterText extends Applet {
    final Font f = new Font("SansSerif", Font.BOLD, 18);

    public void paint(Graphics g) {
        Dimension d = this.getSize();

        g.setColor(Color.white);
        g.fillRect(0, 0, d.width,d.height);
        g.setColor(Color.black);
        g.setFont(f);
        drawCenteredString("This is centered.", d.width, d.height, g);
        g.drawRect(0, 0, d.width-1, d.height-1);
    }

    public void drawCenteredString(String s, int w, int h,
                                   Graphics g) {
        FontMetrics fm = g.getFontMetrics();
        int x = (w - fm.stringWidth(s)) / 2;
        int y = (fm.getAscent() + (h - (fm.getAscent()
            + fm.getDescent())))/2;
        g.drawString(s, x, y);
    }
}
```

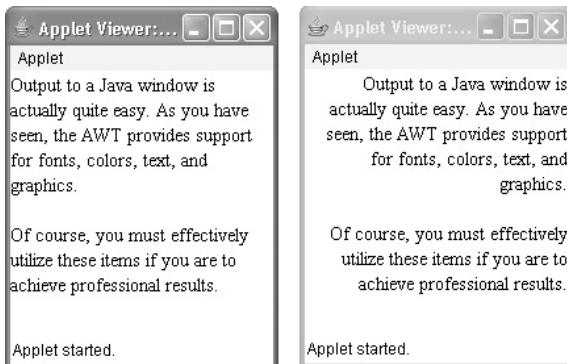
Following is a sample output from this program:



Multiline Text Alignment

When using a word processor, it is common for text to be aligned so that one or more of the edges of the text make a straight line. For example, most word processors can left-justify and/or right-justify text. Most can also center text. In the following program, you will see how to accomplish these actions.

In the program, the string to be justified is broken into individual words. For each word, the program keeps track of its length in the current font and automatically advances to the next line if the word will not fit on the current line. Each completed line is displayed in the window in the currently selected alignment style. Each time you click the mouse in the applet's window, the alignment style is changed. Sample output from this program is shown here:



```
// Demonstrate text alignment.
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;
/* <title>Text Layout</title>
   <applet code="TextLayout" width=200 height=200>
      <param name="text" value="Output to a Java window is actually
         quite easy.
         As you have seen, the AWT provides support for
         fonts, colors, text, and graphics. <P> Of course,
         you must effectively utilize these items
         if you are to achieve professional results.">
      <param name="fontname" value="Serif">
      <param name="fontSize" value="14">
   </applet>
*/
```

```
public class TextLayout extends Applet {
    final int LEFT = 0;
    final int RIGHT = 1;
    final int CENTER = 2;
    final int LEFTRIGHT = 3;
    int align;
```

```
Dimension d;
Font f;
FontMetrics fm;
int fontSize;
int fh, bl;
int space;
String text;

public void init() {
    setBackground(Color.white);
    text = getParameter("text");
    try {
        fontSize = Integer.parseInt(getParameter("fontSize"));
    } catch (NumberFormatException e) {
        fontSize=14;
    }
    align = LEFT;
    addMouseListener(new MyMouseAdapter(this));
}

public void paint(Graphics g) {
    update(g);
}

public void update(Graphics g) {
    d = getSize();
    g.setColor(backgroundColor);
    g.fillRect(0,0,d.width, d.height);
    if(f==null) f = new Font(getParameter("fontname"),
                             Font.PLAIN, fontSize);
    g.setFont(f);
    if(fm == null) {
        fm = g.getFontMetrics();
        bl = fm.getAscent();
        fh = bl + fm.getDescent();
        space = fm.stringWidth(" ");
    }

    g.setColor(Color.black);
    StringTokenizer st = new StringTokenizer(text);
    int x = 0;
    int nextx;
    int y = 0;
    String word, sp;
    int wordCount = 0;
    String line = "";
    while (st.hasMoreTokens()) {
        word = st.nextToken();
        if(word.equals("<P>")) {
            drawString(g, line, wordCount,
                       fm.stringWidth(line), y+bl);
            line = "";
            wordCount = 0;
```

```
x = 0;
y = y + (fh * 2);
}
else {
    int w = fm.stringWidth(word);
    if(( nextx = (x+space+w)) > d.width ) {
        drawString(g, line, wordCount,
                   fm.stringWidth(line), y+bl);
        line = "";
        wordCount = 0;
        x = 0;
        y = y + fh;
    }
    if(x!=0) {sp = " ";} else {sp = "";}
    line = line + sp + word;
    x = x + space + w;
    wordCount++;
}
}
drawString(g, line, wordCount, fm.stringWidth(line), y+bl);
}

public void drawString(Graphics g, String line,
                      int wc, int lineW, int y) {
    switch(align) {
        case LEFT: g.drawString(line, 0, y);
                    break;
        case RIGHT: g.drawString(line, d.width-lineW,y);
                     break;
        case CENTER: g.drawString(line, (d.width-lineW)/2, y);
                     break;
        case LEFTRIGHT:
            if(lineW < (int)(d.width*.75)) {
                g.drawString(line, 0, y);
            }
            else {
                int toFill = (d.width - lineW)/wc;
                int nudge = d.width - lineW - (toFill*wc);
                int s = fm.stringWidth(" ");
                StringTokenizer st = new StringTokenizer(line);
                int x = 0;
                while(st.hasMoreTokens()) {
                    String word = st.nextToken();
                    g.drawString(word, x, y);
                    if(nudge>0) {
                        x = x + fm.stringWidth(word) + space + toFill + 1;
                        nudge--;
                    } else {
                        x = x + fm.stringWidth(word) + space + toFill;
                    }
                }
            }
            break;
    }
}
```

```
        }
    }
}

class MyMouseAdapter extends MouseAdapter {
    TextLayout tl;

    public MyMouseAdapter(TextLayout tl) {
        this.tl = tl;
    }

    public void mouseClicked(MouseEvent me) {
        tl.align = (tl.align + 1) % 4;
        tl.repaint();
    }
}
```

Let's take a closer look at how this applet works. The applet first creates several constants that will be used to determine the alignment style, and then declares several variables. The `init()` method obtains the text that will be displayed. It then initializes the font size in a `try-catch` block, which will set the font size to 14 if the `fontSize` parameter is missing from the HTML. The `text` parameter is a long string of text, with the HTML tag `<P>` as a paragraph separator.

The `update()` method is the engine for this example. It sets the font and gets the baseline and font height from a font metrics object. Next, it creates a `StringTokenizer` and uses it to retrieve the next token (a string separated by whitespace) from the string specified by `text`. If the next token is `<P>`, it advances the vertical spacing. Otherwise, `update()` checks to see if the length of this token in the current font will go beyond the width of the column. If the line is full of text or if there are no more tokens, the line is output by a custom version of `drawString()`.

The first three cases in `drawString()` are simple. Each aligns the string that is passed in `line` to the left or right edge or to the center of the column, depending upon the alignment style. The `LEFTRIGHT` case aligns both the left and right sides of the string. This means that we need to calculate the remaining whitespace (the difference between the width of the string and the width of the column) and distribute that space between each of the words. The last method in this class advances the alignment style each time you click the mouse on the applet's window.

CHAPTER

26

Using AWT Controls, Layout Managers, and Menus

This chapter continues our overview of the Abstract Window Toolkit (AWT). It begins with a look at several of the AWT's controls and layout managers. It then discusses menus and the menu bar. The chapter also includes a discussion of two high-level components: the dialog box and the file dialog box. It concludes with another look at event handling.

Controls are components that allow a user to interact with your application in various ways—for example, a commonly used control is the push button. A *layout manager* automatically positions components within a container. Thus, the appearance of a window is determined by a combination of the controls that it contains and the layout manager used to position them.

In addition to the controls, a frame window can also include a standard-style *menu bar*. Each entry in a menu bar activates a drop-down menu of options from which the user can choose. This constitutes the *main menu* of an application. As a general rule, a menu bar is positioned at the top of a window. Although different in appearance, menu bars are handled in much the same way as are the other controls.

While it is possible to manually position components within a window, doing so is quite tedious. The layout manager automates this task. For the first part of this chapter, which introduces various controls, the default layout manager will be used. This displays components in a container using left-to-right, top-to-bottom organization. Once the controls have been covered, several layout managers will be examined. There, you will see ways to better manage the positioning of controls.

Before continuing, it is important to emphasize that today you will seldom create GUIs based solely on the AWT because more powerful GUI frameworks (Swing and JavaFX) have been developed for Java. However, the material presented here remains important for the following reasons. First, much of the information and many of the techniques related to controls and event handling are generalizable to the other Java GUI frameworks. (As mentioned in the previous chapter, Swing is built upon the AWT.) Second, the layout managers described here can also be used by Swing. Third, for some small applications, the AWT components might be the appropriate choice. Finally, and perhaps most importantly, you may need to maintain or upgrade legacy code that uses the AWT. Therefore, a basic understanding of the AWT is important for all Java programmers.

AWT Control Fundamentals

The AWT supports the following types of controls:

- Labels
- Push buttons
- Check boxes
- Choice lists
- Lists
- Scroll bars
- Text Editing

These controls are subclasses of **Component**. Although this is not a particularly rich set of controls, it is sufficient for simple applications. (Note that both Swing and JavaFX provide a substantially larger, more sophisticated set of controls.)

Adding and Removing Controls

To include a control in a window, you must add it to the window. To do this, you must first create an instance of the desired control and then add it to a window by calling **add()**, which is defined by **Container**. The **add()** method has several forms. The following form is the one that is used for the first part of this chapter:

```
Component add(Component compRef)
```

Here, *compRef* is a reference to an instance of the control that you want to add. A reference to the object is returned. Once a control has been added, it will automatically be visible whenever its parent window is displayed.

Sometimes you will want to remove a control from a window when the control is no longer needed. To do this, call **remove()**. This method is also defined by **Container**. Here is one of its forms:

```
void remove(Component compRef)
```

Here, *compRef* is a reference to the control you want to remove. You can remove all controls by calling **removeAll()**.

Responding to Controls

Except for labels, which are passive, all other controls generate events when they are accessed by the user. For example, when the user clicks on a push button, an event is sent that identifies the push button. In general, your program simply implements the appropriate interface and then registers an event listener for each control that you need to monitor. As explained in Chapter 24, once a listener has been installed, events are automatically sent to it. In the sections that follow, the appropriate interface for each control is specified.

The HeadlessException

Most of the AWT controls described in this chapter have constructors that can throw a **HeadlessException** when an attempt is made to instantiate a GUI component in a non-interactive environment (such as one in which no display, mouse, or keyboard is present). You can use this exception to write code that can adapt to non-interactive environments. (Of course, this is not always possible.) This exception is not handled by the programs in this chapter because an interactive environment is required to demonstrate the AWT controls.

Labels

The easiest control to use is a label. A *label* is an object of type **Label**, and it contains a string, which it displays. Labels are passive controls that do not support any interaction with the user. **Label** defines the following constructors:

```
Label( ) throws HeadlessException  
Label(String str) throws HeadlessException  
Label(String str, int how) throws HeadlessException
```

The first version creates a blank label. The second version creates a label that contains the string specified by *str*. This string is left-justified. The third version creates a label that contains the string specified by *str* using the alignment specified by *how*. The value of *how* must be one of these three constants: **Label.LEFT**, **Label.RIGHT**, or **Label.CENTER**.

You can set or change the text in a label by using the **setText()** method. You can obtain the current label by calling **getText()**. These methods are shown here:

```
void setText(String str)  
String getText( )
```

For **setText()**, *str* specifies the new label. For **getText()**, the current label is returned.

You can set the alignment of the string within the label by calling **setAlignment()**. To obtain the current alignment, call **getAlignment()**. The methods are as follows:

```
void setAlignment(int how)  
int getAlignment( )
```

Here, *how* must be one of the alignment constants shown earlier.

The following example creates three labels and adds them to an applet window:

```
// Demonstrate Labels  
import java.awt.*;  
import java.applet.*;  
/*  
<applet code="LabelDemo" width=300 height=200>  
</applet>  
*/  
  
public class LabelDemo extends Applet {  
    public void init() {  
        Label one = new Label("One");  
        Label two = new Label("Two");  
        Label three = new Label("Three");  
    }
```

```

    // add labels to applet window
    add(one);
    add(two);
    add(three);
}
}

```

Here is sample output from the **LabelDemo** applet. Notice that the labels are organized in the window by the default layout manager. Later, you will see how to control more precisely the placement of the labels.



Using Buttons

Perhaps the most widely used control is the push button. A *push button* is a component that contains a label and that generates an event when it is pressed. Push buttons are objects of type **Button**. **Button** defines these two constructors:

```

Button( ) throws HeadlessException
Button(String str) throws HeadlessException

```

The first version creates an empty button. The second creates a button that contains *str* as a label.

After a button has been created, you can set its label by calling **setLabel()**. You can retrieve its label by calling **getLabel()**. These methods are as follows:

```

void setLabel(String str)
String getLabel( )

```

Here, *str* becomes the new label for the button.

Handling Buttons

Each time a button is pressed, an action event is generated. This is sent to any listeners that previously registered an interest in receiving action event notifications from that component. Each listener implements the **ActionListener** interface. That interface defines the **actionPerformed()** method, which is called when an event occurs. An **ActionEvent** object is supplied as the argument to this method. It contains both a reference to the button that generated the event and a reference to the *action command string* associated with the button. By default, the action command string is the label of the button. Either the button reference or the action command string can be used to identify the button. (You will soon see examples of each approach.)

Here is an example that creates three buttons labeled "Yes", "No", and "Undecided". Each time one is pressed, a message is displayed that reports which button has been pressed. In this version, the action command of the button (which, by default, is its label) is used to determine which button has been pressed. The label is obtained by calling the **getActionCommand()** method on the **ActionEvent** object passed to **actionPerformed()**.

```
// Demonstrate Buttons
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="ButtonDemo" width=250 height=150>
</applet>
*/
public class ButtonDemo extends Applet implements ActionListener {
    String msg = "";
    Button yes, no, maybe;

    public void init() {
        yes = new Button("Yes");
        no = new Button("No");
        maybe = new Button("Undecided");

        add(yes);
        add(no);
        add(maybe);

        yes.addActionListener(this);
        no.addActionListener(this);
        maybe.addActionListener(this);
    }

    public void actionPerformed(ActionEvent ae) {
        String str = ae.getActionCommand();

        if(str.equals("Yes")) {
            msg = "You pressed Yes.";
        }
        else if(str.equals("No")) {
            msg = "You pressed No.";
        }
        else {
            msg = "You pressed Undecided.";
        }

        repaint();
    }

    public void paint(Graphics g) {
        g.drawString(msg, 6, 100);
    }
}
```

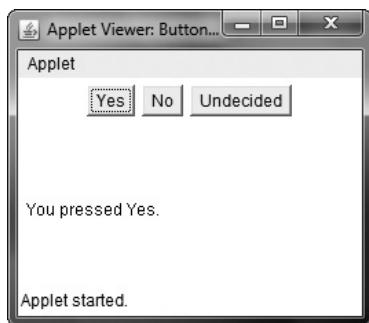


Figure 26-1 Sample output from the **ButtonDemo** applet

Sample output from the **ButtonDemo** program is shown in Figure 26-1.

As mentioned, in addition to comparing button action command strings, you can also determine which button has been pressed by comparing the object obtained from the **getSource()** method to the button objects that you added to the window. To do this, you must keep a list of the objects when they are added. The following applet shows this approach:

```
// Recognize Button objects.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="ButtonList" width=250 height=150>
</applet>
*/
public class ButtonList extends Applet implements ActionListener {
    String msg = "";
    Button bList[] = new Button[3];

    public void init() {
        Button yes = new Button("Yes");
        Button no = new Button("No");
        Button maybe = new Button("Undecided");

        // store references to buttons as added
        bList[0] = (Button) add(yes);
        bList[1] = (Button) add(no);
        bList[2] = (Button) add(maybe);

        // register to receive action events
        for(int i = 0; i < 3; i++) {
            bList[i].addActionListener(this);
        }
    }
}
```

```

public void actionPerformed(ActionEvent ae) {
    for(int i = 0; i < 3; i++) {
        if(ae.getSource() == bList[i]) {
            msg = "You pressed " + bList[i].getLabel();
        }
    }
    repaint();
}

public void paint(Graphics g) {
    g.drawString(msg, 6, 100);
}
}

```

In this version, the program stores each button reference in an array when the buttons are added to the applet window. (Recall that the `add()` method returns a reference to the button when it is added.) Inside `actionPerformed()`, this array is then used to determine which button has been pressed.

For simple programs, it is usually easier to recognize buttons by their labels. However, in situations in which you will be changing the label inside a button during the execution of your program, or using buttons that have the same label, it may be easier to determine which button has been pushed by using its object reference. It is also possible to set the action command string associated with a button to something other than its label by calling `setActionCommand()`. This method changes the action command string, but does not affect the string used to label the button. Thus, setting the action command enables the action command and the label of a button to differ.

In some cases, you can handle the action events generated by a button (or some other type of control) by use of an anonymous inner class (as described in Chapter 24) or a lambda expression (discussed in Chapter 15). For example, assuming the previous programs, here is a set of action event handlers that use lambda expressions:

```

// Use lambda expressions to handle action events.
yes.addActionListener((ae) -> {
    msg = "You pressed " + ae.getActionCommand();
    repaint();
});

no.addActionListener((ae) -> {
    msg = "You pressed " + ae.getActionCommand();
    repaint();
});

maybe.addActionListener((ae) -> {
    msg = "You pressed " + ae.getActionCommand();
    repaint();
});

```

This code works because `ActionListener` defines a functional interface, which is an interface with exactly one abstract method. Thus, it can be used by a lambda expression. In general, you can use a lambda expression to handle an AWT event when its listener defines a functional interface. For example, `ItemListener` is also a functional interface. Of course,

whether you use the traditional approach, an anonymous inner class, or a lambda expression will be determined by the precise nature of your application. The remaining examples in this chapter use the traditional approach to event handling so that they can be compiled by nearly any version of Java. However, you might find it interesting to try converting the event handlers to lambda expressions or anonymous inner classes, where appropriate.

Applying Check Boxes

A *checkbox* is a control that is used to turn an option on or off. It consists of a small box that can either contain a check mark or not. There is a label associated with each checkbox that describes what option the box represents. You change the state of a checkbox by clicking on it. Check boxes can be used individually or as part of a group. Check boxes are objects of the **Checkbox** class.

Checkbox supports these constructors:

```
Checkbox( ) throws HeadlessException  
Checkbox(String str) throws HeadlessException  
Checkbox(String str, boolean on) throws HeadlessException  
Checkbox(String str, boolean on, CheckboxGroup cbGroup) throws HeadlessException  
Checkbox(String str, CheckboxGroup cbGroup, boolean on) throws HeadlessException
```

The first form creates a checkbox whose label is initially blank. The state of the checkbox is unchecked. The second form creates a checkbox whose label is specified by *str*. The state of the checkbox is unchecked. The third form allows you to set the initial state of the checkbox. If *on* is **true**, the checkbox is initially checked; otherwise, it is cleared. The fourth and fifth forms create a checkbox whose label is specified by *str* and whose group is specified by *cbGroup*. If this checkbox is not part of a group, then *cbGroup* must be **null**. (Check box groups are described in the next section.) The value of *on* determines the initial state of the checkbox.

To retrieve the current state of a checkbox, call **getState()**. To set its state, call **setState()**. You can obtain the current label associated with a checkbox by calling **getLabel()**. To set the label, call **setLabel()**. These methods are as follows:

```
boolean getState()  
void setState(boolean on)  
String getLabel()  
void setLabel(String str)
```

Here, if *on* is **true**, the box is checked. If it is **false**, the box is cleared. The string passed in *str* becomes the new label associated with the invoking checkbox.

Handling Check Boxes

Each time a checkbox is selected or deselected, an item event is generated. This is sent to any listeners that previously registered an interest in receiving item event notifications from that component. Each listener implements the **ItemListener** interface. That interface defines the **itemStateChanged()** method. An **ItemEvent** object is supplied as the argument to this method. It contains information about the event (for example, whether it was a selection or deselection).

The following program creates four check boxes. The initial state of the first box is checked. The status of each check box is displayed. Each time you change the state of a check box, the status display is updated.

```
// Demonstrate check boxes.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="CheckboxDemo" width=240 height=200>
</applet>
*/
public class CheckboxDemo extends Applet implements ItemListener {
    String msg = "";
    Checkbox windows, android, solaris, mac;

    public void init() {
        windows = new Checkbox("Windows", null, true);
        android = new Checkbox("Android");
        solaris = new Checkbox("Solaris");
        mac = new Checkbox("Mac OS");

        add(windows);
        add(android);
        add(solaris);
        add(mac);

        windows.addItemListener(this);
        android.addItemListener(this);
        solaris.addItemListener(this);
        mac.addItemListener(this);
    }

    public void itemStateChanged(ItemEvent ie) {
        repaint();
    }

    // Display current state of the check boxes.
    public void paint(Graphics g) {
        msg = "Current state: ";
        g.drawString(msg, 6, 80);
        msg = " Windows: " + windows.getState();
        g.drawString(msg, 6, 100);
        msg = " Android: " + android.getState();
        g.drawString(msg, 6, 120);
        msg = " Solaris: " + solaris.getState();
        g.drawString(msg, 6, 140);
        msg = " Mac OS: " + mac.getState();
        g.drawString(msg, 6, 160);
    }
}
```

Sample output is shown in Figure 26-2.



Figure 26-2 Sample output from the **CheckboxDemo** applet

CheckboxGroup

It is possible to create a set of mutually exclusive check boxes in which one and only one check box in the group can be checked at any one time. These check boxes are often called *radio buttons*, because they act like the station selector on a car radio—only one station can be selected at any one time. To create a set of mutually exclusive check boxes, you must first define the group to which they will belong and then specify that group when you construct the check boxes. Check box groups are objects of type **CheckboxGroup**. Only the default constructor is defined, which creates an empty group.

You can determine which check box in a group is currently selected by calling **getSelectedCheckbox()**. You can set a check box by calling **setSelectedCheckbox()**. These methods are as follows:

```
Checkbox getSelectedCheckbox()
void setSelectedCheckbox(Checkbox which)
```

Here, *which* is the check box that you want to be selected. The previously selected check box will be turned off.

Here is a program that uses check boxes that are part of a group:

```
// Demonstrate check box group.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="CBGroup" width=240 height=200>
</applet>
*/
public class CBGroup extends Applet implements ItemListener {
    String msg = "";
    Checkbox windows, android, solaris, mac;
    CheckboxGroup cbg;
```

```
public void init() {  
    cbg = new CheckboxGroup();  
    windows = new Checkbox("Windows", cbg, true);  
    android = new Checkbox("Android", cbg, false);  
    solaris = new Checkbox("Solaris", cbg, false);  
    mac = new Checkbox("Mac OS", cbg, false);  
  
    add(windows);  
    add(android);  
    add(solaris);  
    add(mac);  
  
    windows.addItemListener(this);  
    android.addItemListener(this);  
    solaris.addItemListener(this);  
    mac.addItemListener(this);  
}  
  
public void itemStateChanged(ItemEvent ie) {  
    repaint();  
}  
  
// Display current state of the check boxes.  
public void paint(Graphics g) {  
    msg = "Current selection: ";  
    msg += cbg.getSelectedCheckbox().getLabel();  
    g.drawString(msg, 6, 100);  
}  
}
```

Sample output generated by the **CBGroup** applet is shown in Figure 26-3. Notice that the check boxes are now circular in shape.



Figure 26-3 Sample output from the **CBGroup** applet

Choice Controls

The **Choice** class is used to create a *pop-up list* of items from which the user may choose. Thus, a **Choice** control is a form of menu. When inactive, a **Choice** component takes up only enough space to show the currently selected item. When the user clicks on it, the whole list of choices pops up, and a new selection can be made. Each item in the list is a string that appears as a left-justified label in the order it is added to the **Choice** object. **Choice** defines only the default constructor, which creates an empty list.

To add a selection to the list, call **add()**. It has this general form:

```
void add(String name)
```

Here, *name* is the name of the item being added. Items are added to the list in the order in which calls to **add()** occur.

To determine which item is currently selected, you may call either **getSelectedItem()** or **getSelectedIndex()**. These methods are shown here:

```
String getSelectedItem()
int getSelectedIndex()
```

The **getSelectedItem()** method returns a string containing the name of the item.

getSelectedIndex() returns the index of the item. The first item is at index 0. By default, the first item added to the list is selected.

To obtain the number of items in the list, call **getItemCount()**. You can set the currently selected item using the **select()** method with either a zero-based integer index or a string that will match a name in the list. These methods are shown here:

```
int getItemCount()
void select(int index)
void select(String name)
```

Given an index, you can obtain the name associated with the item at that index by calling **getItem()**, which has this general form:

```
String getItem(int index)
```

Here, *index* specifies the index of the desired item.

Handling Choice Lists

Each time a choice is selected, an item event is generated. This is sent to any listeners that previously registered an interest in receiving item event notifications from that component. Each listener implements the **ItemListener** interface. That interface defines the **itemStateChanged()** method. An **ItemEvent** object is supplied as the argument to this method.

Here is an example that creates two **Choice** menus. One selects the operating system. The other selects the browser.

```
// Demonstrate Choice lists.
import java.awt.*;
import java.awt.event.*;
```

```
import java.applet.*;
/*
<applet code="ChoiceDemo" width=300 height=180>
</applet>
*/
public class ChoiceDemo extends Applet implements ItemListener {
    Choice os, browser;
    String msg = "";
    public void init() {
        os = new Choice();
        browser = new Choice();

        // add items to os list
        os.add("Windows");
        os.add("Android");
        os.add("Solaris");
        os.add("Mac OS");

        // add items to browser list
        browser.add("Internet Explorer");
        browser.add("Firefox");
        browser.add("Chrome");

        // add choice lists to window
        add(os);
        add(browser);

        // register to receive item events
        os.addItemListener(this);
        browser.addItemListener(this);
    }
    public void itemStateChanged(ItemEvent ie) {
        repaint();
    }
    // Display current selections.
    public void paint(Graphics g) {
        msg = "Current OS: ";
        msg += os.getSelectedItem();
        g.drawString(msg, 6, 120);
        msg = "Current Browser: ";
        msg += browser.getSelectedItem();
        g.drawString(msg, 6, 140);
    }
}
```

Sample output is shown in Figure 26-4.

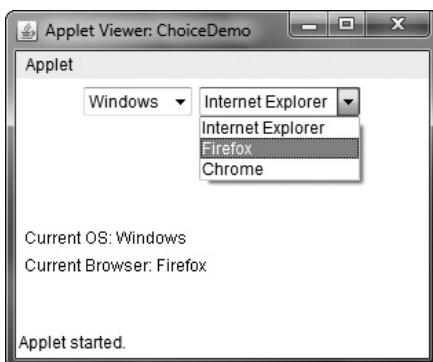


Figure 26-4 Sample output from the **ChoiceDemo** applet

Using Lists

The **List** class provides a compact, multiple-choice, scrolling selection list. Unlike the **Choice** object, which shows only the single selected item in the menu, a **List** object can be constructed to show any number of choices in the visible window. It can also be created to allow multiple selections. **List** provides these constructors:

```
List( ) throws HeadlessException
List(int numRows) throws HeadlessException
List(int numRows, boolean multipleSelect) throws HeadlessException
```

The first version creates a **List** control that allows only one item to be selected at any one time. In the second form, the value of *numRows* specifies the number of entries in the list that will always be visible (others can be scrolled into view as needed). In the third form, if *multipleSelect* is **true**, then the user may select two or more items at a time. If it is **false**, then only one item may be selected.

To add a selection to the list, call **add()**. It has the following two forms:

```
void add(String name)
void add(String name, int index)
```

Here, *name* is the name of the item added to the list. The first form adds items to the end of the list. The second form adds the item at the index specified by *index*. Indexing begins at zero. You can specify **-1** to add the item to the end of the list.

For lists that allow only single selection, you can determine which item is currently selected by calling either **getSelectedItem()** or **getSelectedIndex()**. These methods are shown here:

```
String getSelectedItem( )
int getSelectedIndex( )
```

The **getSelectedItem()** method returns a string containing the name of the item. If more than one item is selected, or if no selection has yet been made, **null** is returned. **getSelectedIndex()** returns the index of the item. The first item is at index 0. If more than one item is selected, or if no selection has yet been made, **-1** is returned.

For lists that allow multiple selection, you must use either `getSelectedItems()` or `getSelectedIndexes()`, shown here, to determine the current selections:

```
String[ ] getSelectedItems( )  
int[ ] getSelectedIndexes( )
```

`getSelectedItems()` returns an array containing the names of the currently selected items. `getSelectedIndexes()` returns an array containing the indexes of the currently selected items.

To obtain the number of items in the list, call `getItemCount()`. You can set the currently selected item by using the `select()` method with a zero-based integer index. These methods are shown here:

```
int getItemCount( )  
void select(int index)
```

Given an index, you can obtain the name associated with the item at that index by calling `getItem()`, which has this general form:

String getItem(int *index*)

Here, *index* specifies the index of the desired item.

Handling Lists

To process list events, you will need to implement the **ActionListener** interface. Each time a **List** item is double-clicked, an **ActionEvent** object is generated. Its **getActionCommand()** method can be used to retrieve the name of the newly selected item. Also, each time an item is selected or deselected with a single click, an **ItemEvent** object is generated. Its **getStateChange()** method can be used to determine whether a selection or deselection triggered this event. **getItemSelectable()** returns a reference to the object that triggered this event.

Here is an example that converts the **Choice** controls in the preceding section into **List** components, one multiple choice and the other single choice:

```
os.add("Android");
os.add("Solaris");
os.add("Mac OS");

// add items to browser list
browser.add("Internet Explorer");
browser.add("Firefox");
browser.add("Chrome");

browser.select(1);

// add lists to window
add(os);
add(browser);

// register to receive action events
os.addActionListener(this);
browser.addActionListener(this);
}

public void actionPerformed(ActionEvent ae) {
    repaint();
}

// Display current selections.
public void paint(Graphics g) {
    int idx[];

    msg = "Current OS: ";
    idx = os.getSelectedIndexes();
    for(int i=0; i<idx.length; i++)
        msg += os.getItem(idx[i]) + " ";
    g.drawString(msg, 6, 120);
    msg = "Current Browser: ";
    msg += browser.getSelectedItem();
    g.drawString(msg, 6, 140);
}
}
```

Sample output generated by the **ListDemo** applet is shown in Figure 26-5.

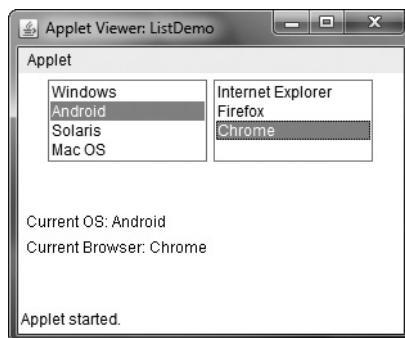


Figure 26-5 Sample output from the **ListDemo** applet

Managing Scroll Bars

Scroll bars are used to select continuous values between a specified minimum and maximum. Scroll bars may be oriented horizontally or vertically. A scroll bar is actually a composite of several individual parts. Each end has an arrow that you can click to move the current value of the scroll bar one unit in the direction of the arrow. The current value of the scroll bar relative to its minimum and maximum values is indicated by the *slider box* (or *thumb*) for the scroll bar. The slider box can be dragged by the user to a new position. The scroll bar will then reflect this value. In the background space on either side of the thumb, the user can click to cause the thumb to jump in that direction by some increment larger than 1. Typically, this action translates into some form of page up and page down. Scroll bars are encapsulated by the **Scrollbar** class.

Scrollbar defines the following constructors:

```
Scrollbar( ) throws HeadlessException
Scrollbar(int style) throws HeadlessException
Scrollbar(int style, int initialValue, int thumbSize, int min, int max)
    throws HeadlessException
```

The first form creates a vertical scroll bar. The second and third forms allow you to specify the orientation of the scroll bar. If *style* is **Scrollbar.VERTICAL**, a vertical scroll bar is created. If *style* is **Scrollbar.HORIZONTAL**, the scroll bar is horizontal. In the third form of the constructor, the initial value of the scroll bar is passed in *initialValue*. The number of units represented by the height of the thumb is passed in *thumbSize*. The minimum and maximum values for the scroll bar are specified by *min* and *max*.

If you construct a scroll bar by using one of the first two constructors, then you need to set its parameters by using **setValues()**, shown here, before it can be used:

```
void setValues(int initialValue, int thumbSize, int min, int max)
```

The parameters have the same meaning as they have in the third constructor just described.

To obtain the current value of the scroll bar, call **getValue()**. It returns the current setting. To set the current value, call **setValue()**. These methods are as follows:

```
int getValue( )
void setValue(int newValue)
```

Here, *newValue* specifies the new value for the scroll bar. When you set a value, the slider box inside the scroll bar will be positioned to reflect the new value.

You can also retrieve the minimum and maximum values via **getMinimum()** and **getMaximum()**, shown here:

```
int getMinimum( )
int getMaximum( )
```

They return the requested quantity.

By default, 1 is the increment added to or subtracted from the scroll bar each time it is scrolled up or down one line. You can change this increment by calling **setUnitIncrement()**. By default, page-up and page-down increments are 10. You can change this value by calling **setBlockIncrement()**. These methods are shown here:

```
void setUnitIncrement(int newIncr)
void setBlockIncrement(int newIncr)
```

Handling Scroll Bars

To process scroll bar events, you need to implement the **AdjustmentListener** interface. Each time a user interacts with a scroll bar, an **AdjustmentEvent** object is generated. Its **getAdjustmentType()** method can be used to determine the type of the adjustment. The types of adjustment events are as follows:

BLOCK_DECREMENT	A page-down event has been generated.
BLOCK_INCREMENT	A page-up event has been generated.
TRACK	An absolute tracking event has been generated.
UNIT_DECREMENT	The line-down button in a scroll bar has been pressed.
UNIT_INCREMENT	The line-up button in a scroll bar has been pressed.

The following example creates both a vertical and a horizontal scroll bar. The current settings of the scroll bars are displayed. If you drag the mouse while inside the window, the coordinates of each drag event are used to update the scroll bars. An asterisk is displayed at the current drag position. Notice the use of **setPreferredSize()** to set the size of the scrollbars.

```
// Demonstrate scroll bars.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="SBDemo" width=300 height=200>
</applet>
*/
public class SBDemo extends Applet
    implements AdjustmentListener, MouseMotionListener {
    String msg = "";
    Scrollbar vertSB, horzSB;

    public void init() {
        int width = Integer.parseInt(getParameter("width"));
        int height = Integer.parseInt(getParameter("height"));

        vertSB = new Scrollbar(Scrollbar.VERTICAL,
            0, 1, 0, height);
        vertSB.setPreferredSize(new Dimension(20, 100));

        horzSB = new Scrollbar(Scrollbar.HORIZONTAL,
            0, 1, 0, width);
        horzSB.setPreferredSize(new Dimension(100, 20));

        add(vertSB);
        add(horzSB);

        // register to receive adjustment events
    }
}
```

```
vertSB.addAdjustmentListener(this);
horzSB.addAdjustmentListener(this);

    addMouseMotionListener(this);
}

public void adjustmentValueChanged(AdjustmentEvent ae) {
    repaint();
}

// Update scroll bars to reflect mouse dragging.
public void mouseDragged(MouseEvent me) {
    int x = me.getX();
    int y = me.getY();
    vertSB.setValue(y);
    horzSB.setValue(x);
    repaint();
}

// Necessary for MouseMotionListener
public void mouseMoved(MouseEvent me) {

}

// Display current value of scroll bars.
public void paint(Graphics g) {
    msg = "Vertical: " + vertSB.getValue();
    msg += ", Horizontal: " + horzSB.getValue();
    g.drawString(msg, 6, 160);

    // show current mouse drag position
    g.drawString("*", horzSB.getValue(),
                vertSB.getValue());
}
}
```

Sample output from the **SBDemo** applet is shown in Figure 26-6.

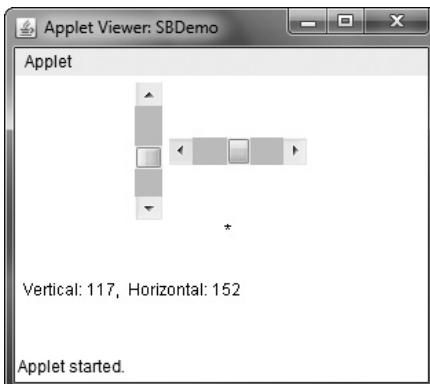


Figure 26-6 Sample output from the **SBDemo** applet

Using a **TextField**

The **TextField** class implements a single-line text-entry area, usually called an *edit control*. Text fields allow the user to enter strings and to edit the text using the arrow keys, cut and paste keys, and mouse selections. **TextField** is a subclass of **TextComponent**. **TextField** defines the following constructors:

```
TextField( ) throws HeadlessException
TextField(int numChars) throws HeadlessException
TextField(String str) throws HeadlessException
TextField(String str, int numChars) throws HeadlessException
```

The first version creates a default text field. The second form creates a text field that is *numChars* characters wide. The third form initializes the text field with the string contained in *str*. The fourth form initializes a text field and sets its width.

TextField (and its superclass **TextComponent**) provides several methods that allow you to utilize a text field. To obtain the string currently contained in the text field, call **getText()**. To set the text, call **setText()**. These methods are as follows:

```
String getText( )
void setText(String str)
```

Here, *str* is the new string.

The user can select a portion of the text in a text field. Also, you can select a portion of text under program control by using **select()**. Your program can obtain the currently selected text by calling **getSelectedText()**. These methods are shown here:

```
String getSelectedText( )
void select(int startIndex, int endIndex)
```

getSelectedText() returns the selected text. The **select()** method selects the characters beginning at *startIndex* and ending at *endIndex* –1.

You can control whether the contents of a text field may be modified by the user by calling **setEditable()**. You can determine editability by calling **isEditable()**. These methods are shown here:

```
boolean isEditable( )
void setEditable(boolean canEdit)
```

isEditable() returns **true** if the text may be changed and **false** if not. In **setEditable()**, if *canEdit* is **true**, the text may be changed. If it is **false**, the text cannot be altered.

There may be times when you will want the user to enter text that is not displayed, such as a password. You can disable the echoing of the characters as they are typed by calling **setEchoChar()**. This method specifies a single character that the **TextField** will display when characters are entered (thus, the actual characters typed will not be shown). You can check a text field to see if it is in this mode with the **echoCharIsSet()** method. You can retrieve the echo character by calling the **getEchoChar()** method. These methods are as follows:

```
void setEchoChar(char ch)
boolean echoCharIsSet( )
char getEchoChar( )
```

Here, *ch* specifies the character to be echoed. If *ch* is zero, then normal echoing is restored.

Handling a TextField

Since text fields perform their own editing functions, your program generally will not respond to individual key events that occur within a text field. However, you may want to respond when the user presses ENTER. When this occurs, an action event is generated.

Here is an example that creates the classic user name and password screen:

```
// Demonstrate text field.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="TextFieldDemo" width=380 height=150>
</applet>
*/
public class TextFieldDemo extends Applet
    implements ActionListener {

    TextField name, pass;

    public void init() {
        Label namep = new Label("Name: ", Label.RIGHT);
        Label passp = new Label("Password: ", Label.RIGHT);
        name = new TextField(12);
        pass = new TextField(8);
        pass.setEchoChar('?');

        add(namep);
        add(name);
        add(passp);
        add(pass);

        // register to receive action events
        name.addActionListener(this);
        pass.addActionListener(this);
    }

    // User pressed Enter.
    public void actionPerformed(ActionEvent ae) {
        repaint();
    }

    public void paint(Graphics g) {
        g.drawString("Name: " + name.getText(), 6, 60);
        g.drawString("Selected text in name: "
                    + name.getSelectedText(), 6, 80);
        g.drawString("Password: " + pass.getText(), 6, 100);
    }
}
```

Sample output from the **TextFieldDemo** applet is shown in Figure 26-7.



Figure 26-7 Sample output from the **TextFieldDemo** applet

Using a **TextArea**

Sometimes a single line of text input is not enough for a given task. To handle these situations, the AWT includes a simple multiline editor called **TextArea**. Following are the constructors for **TextArea**:

```
TextArea( ) throws HeadlessException
TextArea(int numLines, int numChars) throws HeadlessException
TextArea(String str) throws HeadlessException
TextArea(String str, int numLines, int numChars) throws HeadlessException
TextArea(String str, int numLines, int numChars, int sBars) throws HeadlessException
```

Here, *numLines* specifies the height, in lines, of the text area, and *numChars* specifies its width, in characters. Initial text can be specified by *str*. In the fifth form, you can specify the scroll bars that you want the control to have. *sBars* must be one of these values:

SCROLLBARS_BOTH	SCROLLBARS_NONE
SCROLLBARS_HORIZONTAL_ONLY	SCROLLBARS_VERTICAL_ONLY

TextArea is a subclass of **TextComponent**. Therefore, it supports the **getText()**, **setText()**, **getSelectedText()**, **select()**, **isEditable()**, and **setEditable()** methods described in the preceding section.

TextArea adds the following editing methods:

```
void append(String str)
void insert(String str, int index)
void replaceRange(String str, int startIndex, int endIndex)
```

The **append()** method appends the string specified by *str* to the end of the current text. **insert()** inserts the string passed in *str* at the specified index. To replace text, call **replaceRange()**. It replaces the characters from *startIndex* to *endIndex*-1, with the replacement text passed in *str*.

Text areas are almost self-contained controls. Your program incurs virtually no management overhead. Normally, your program simply obtains the current text when it is needed. You can, however, listen for **TextEvents**, if you choose.

The following program creates a **TextArea** control:

```
// Demonstrate TextArea.
import java.awt.*;
import java.applet.*;
/*
<applet code="TextAreaDemo" width=300 height=250>
</applet>
*/
public class TextAreaDemo extends Applet {
    public void init() {
        String val =
            "Java 8 is the latest version of the most\n" +
            "widely-used computer language for Internet programming.\n" +
            "Building on a rich heritage, Java has advanced both\n" +
            "the art and science of computer language design.\n\n" +
            "One of the reasons for Java's ongoing success is its\n" +
            "constant, steady rate of evolution. Java has never stood\n" +
            "still. Instead, Java has consistently adapted to the\n" +
            "rapidly changing landscape of the networked world.\n" +
            "Moreover, Java has often led the way, charting the\n" +
            "course for others to follow.";

        TextArea text = new TextArea(val, 10, 30);
        add(text);
    }
}
```

Here is sample output from the **TextAreaDemo** applet:



Understanding Layout Managers

All of the components that we have shown so far have been positioned by the default layout manager. As we mentioned at the beginning of this chapter, a layout manager automatically arranges your controls within a window by using some type of algorithm. If you have

programmed for other GUI environments, such as Windows, then you may have laid out your controls by hand. While it is possible to lay out Java controls by hand, too, you generally won't want to, for two main reasons. First, it is very tedious to manually lay out a large number of components. Second, sometimes the width and height information is not yet available when you need to arrange some control, because the native toolkit components haven't been realized. This is a chicken-and-egg situation; it is pretty confusing to figure out when it is okay to use the size of a given component to position it relative to another.

Each **Container** object has a layout manager associated with it. A layout manager is an instance of any class that implements the **LayoutManager** interface. The layout manager is set by the **setLayout()** method. If no call to **setLayout()** is made, then the default layout manager is used. Whenever a container is resized (or sized for the first time), the layout manager is used to position each of the components within it.

The **setLayout()** method has the following general form:

```
void setLayout(LayoutManager layoutObj)
```

Here, *layoutObj* is a reference to the desired layout manager. If you wish to disable the layout manager and position components manually, pass **null** for *layoutObj*. If you do this, you will need to determine the shape and position of each component manually, using the **setBounds()** method defined by **Component**. Normally, you will want to use a layout manager.

Each layout manager keeps track of a list of components that are stored by their names. The layout manager is notified each time you add a component to a container. Whenever the container needs to be resized, the layout manager is consulted via its **minimumLayoutSize()** and **preferredLayoutSize()** methods. Each component that is being managed by a layout manager contains the **getPreferredSize()** and **getMinimumSize()** methods. These return the preferred and minimum size required to display each component. The layout manager will honor these requests if at all possible, while maintaining the integrity of the layout policy. You may override these methods for controls that you subclass. Default values are provided otherwise.

Java has several predefined **LayoutManager** classes, several of which are described next. You can use the layout manager that best fits your application.

FlowLayout

FlowLayout is the default layout manager. This is the layout manager that the preceding examples have used. **FlowLayout** implements a simple layout style, which is similar to how words flow in a text editor. The direction of the layout is governed by the container's component orientation property, which, by default, is left to right, top to bottom. Therefore, by default, components are laid out line-by-line beginning at the upper-left corner. In all cases, when a line is filled, layout advances to the next line. A small space is left between each component, above and below, as well as left and right. Here are the constructors for **FlowLayout**:

```
FlowLayout( )
FlowLayout(int how)
FlowLayout(int how, int horz, int vert)
```

The first form creates the default layout, which centers components and leaves five pixels of space between each component. The second form lets you specify how each line is aligned. Valid values for *how* are as follows:

```
FlowLayout.LEFT  
FlowLayout.CENTER  
FlowLayout.RIGHT  
FlowLayout.LEADING  
FlowLayout.TRAILING
```

These values specify left, center, right, leading edge, and trailing edge alignment, respectively. The third constructor allows you to specify the horizontal and vertical space left between components in *horz* and *vert*, respectively.

Here is a version of the **CheckboxDemo** applet shown earlier in this chapter, modified so that it uses left-aligned flow layout:

```
// Use left-aligned flow layout.  
import java.awt.*;  
import java.awt.event.*;  
import java.applet.*;  
/*  
 <applet code="FlowLayoutDemo" width=240 height=200>  
 </applet>  
*/  
  
public class FlowLayoutDemo extends Applet  
    implements ItemListener {  
  
    String msg = "";  
    Checkbox windows, android, solaris, mac;  
  
    public void init() {  
        // set left-aligned flow layout  
        setLayout(new FlowLayout(FlowLayout.LEFT));  
  
        windows = new Checkbox("Windows", null, true);  
        android = new Checkbox("Android");  
        solaris = new Checkbox("Solaris");  
        mac = new Checkbox("Mac OS");  
  
        add(windows);  
        add(android);  
        add(solaris);  
        add(mac);  
  
        // register to receive item events  
        windows.addItemListener(this);  
        android.addItemListener(this);  
        solaris.addItemListener(this);  
        mac.addItemListener(this);  
    }  
}
```

```

// Repaint when status of a check box changes.
public void itemStateChanged(ItemEvent ie) {
    repaint();
}

// Display current state of the check boxes.
public void paint(Graphics g) {

    msg = "Current state: ";
    g.drawString(msg, 6, 80);
    msg = " Windows: " + windows.getState();
    g.drawString(msg, 6, 100);
    msg = " Android: " + android.getState();
    g.drawString(msg, 6, 120);
    msg = " Solaris: " + solaris.getState();
    g.drawString(msg, 6, 140);
    msg = " Mac: " + mac.getState();
    g.drawString(msg, 6, 160);
}
}

```

Here is sample output generated by the **FlowLayoutDemo** applet. Compare this with the output from the **CheckboxDemo** applet, shown earlier in Figure 26-2.

BorderLayout

The **BorderLayout** class implements a common layout style for top-level windows. It has four narrow, fixed-width components at the edges and one large area in the center. The four sides are referred to as north, south, east, and west. The middle area is called the center. Here are the constructors defined by **BorderLayout**:

```

BorderLayout()
BorderLayout(int horz, int vert)

```

The first form creates a default border layout. The second allows you to specify the horizontal and vertical space left between components in *horz* and *vert*, respectively.

BorderLayout defines the following constants that specify the regions:

BorderLayout.CENTER	BorderLayout.SOUTH
BorderLayout.EAST	BorderLayout.WEST
BorderLayout.NORTH	

When adding components, you will use these constants with the following form of **add()**, which is defined by **Container**:

```
void add(Component compRef, Object region)
```

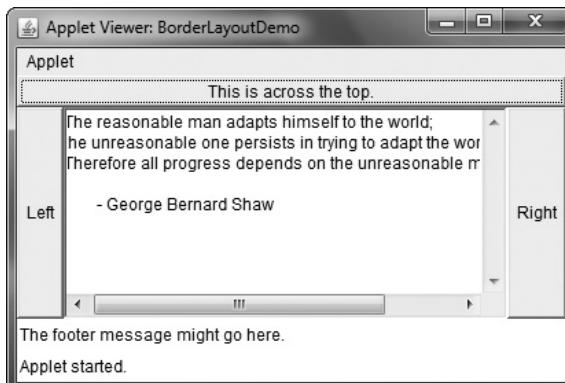
Here, *compRef* is a reference to the component to be added, and *region* specifies where the component will be added.



Here is an example of a **BorderLayout** with a component in each layout area:

```
// Demonstrate BorderLayout.  
import java.awt.*;  
import java.applet.*;  
import java.util.*;  
/*  
<applet code="BorderLayoutDemo" width=400 height=200>  
</applet>  
*/  
  
public class BorderLayoutDemo extends Applet {  
    public void init() {  
        setLayout(new BorderLayout());  
  
        add(new Button("This is across the top."),  
            BorderLayout.NORTH);  
        add(new Label("The footer message might go here."),  
            BorderLayout.SOUTH);  
        add(new Button("Right"), BorderLayout.EAST);  
        add(new Button("Left"), BorderLayout.WEST);  
  
        String msg = "The reasonable man adapts " +  
            "himself to the world;\n" +  
            "the unreasonable one persists in " +  
            "trying to adapt the world to himself.\n" +  
            "Therefore all progress depends " +  
            "on the unreasonable man.\n\n" +  
            "           - George Bernard Shaw\n\n";  
  
        add(new TextArea(msg), BorderLayout.CENTER);  
    }  
}
```

Sample output from the **BorderLayoutDemo** applet is shown here:



Using Insets

Sometimes you will want to leave a small amount of space between the container that holds your components and the window that contains it. To do this, override the `getInsets()` method that is defined by **Container**. This method returns an **Insets** object that contains the top, bottom, left, and right inset to be used when the container is displayed. These values are used by the layout manager to inset the components when it lays out the window. The constructor for **Insets** is shown here:

```
Insets(int top, int left, int bottom, int right)
```

The values passed in *top*, *left*, *bottom*, and *right* specify the amount of space between the container and its enclosing window.

The `getInsets()` method has this general form:

```
Insets getInsets()
```

When overriding this method, you must return a new **Insets** object that contains the inset spacing you desire.

Here is the preceding **BorderLayout** example modified so that it insets its components ten pixels from each border. The background color has been set to cyan to help make the insets more visible.

```
// Demonstrate BorderLayout with insets.
import java.awt.*;
import java.applet.*;
import java.util.*;
/*
<applet code="InsetsDemo" width=400 height=200>
</applet>
*/
public class InsetsDemo extends Applet {
    public void init() {
        // set background color so insets can be easily seen
        setBackground(Color.cyan);

        setLayout(new BorderLayout());

        add(new Button("This is across the top."),
            BorderLayout.NORTH);
        add(new Label("The footer message might go here."),
            BorderLayout.SOUTH);
        add(new Button("Right"), BorderLayout.EAST);
        add(new Button("Left"), BorderLayout.WEST);

        String msg = "The reasonable man adapts " +
            "himself to the world;\n" +
            "the unreasonable one persists in " +
            "trying to adapt the world to himself.\n" +
            "Therefore all progress depends " +
            "on the unreasonable man.\n\n" +
            " - George Bernard Shaw\n\n";
    }
}
```

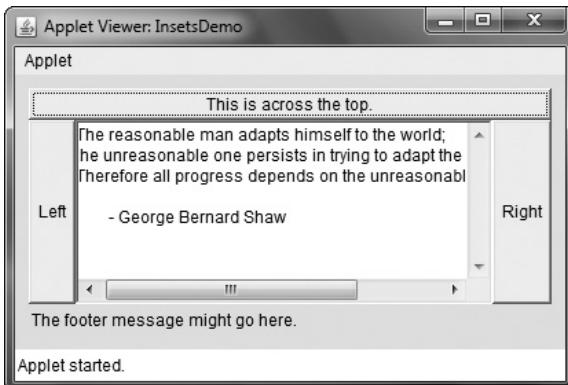
```

        add(new TextArea(msg), BorderLayout.CENTER);
    }

    // add insets
    public Insets getInsets() {
        return new Insets(10, 10, 10, 10);
    }
}

```

Sample output from the **InsetsDemo** applet is shown here:



GridLayout

GridLayout lays out components in a two-dimensional grid. When you instantiate a **GridLayout**, you define the number of rows and columns. The constructors supported by **GridLayout** are shown here:

```

GridLayout()
GridLayout(int numRows, int numColumns)
GridLayout(int numRows, int numColumns, int horz, int vert)

```

The first form creates a single-column grid layout. The second form creates a grid layout with the specified number of rows and columns. The third form allows you to specify the horizontal and vertical space left between components in *horz* and *vert*, respectively. Either *numRows* or *numColumns* can be zero. Specifying *numRows* as zero allows for unlimited-length columns. Specifying *numColumns* as zero allows for unlimited-length rows.

Here is a sample program that creates a 4×4 grid and fills it in with 15 buttons, each labeled with its index:

```

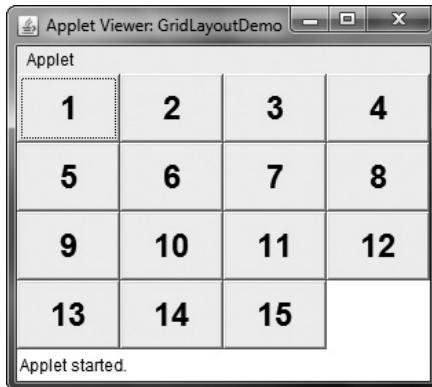
// Demonstrate GridLayout
import java.awt.*;
import java.applet.*;
/*
<applet code="GridLayoutDemo" width=300 height=200>
</applet>
*/

```

```
public class GridLayoutDemo extends Applet {
    static final int n = 4;
    public void init() {
        setLayout(new GridLayout(n, n));
        setFont(new Font("SansSerif", Font.BOLD, 24));

        for(int i = 0; i < n; i++) {
            for(int j = 0; j < n; j++) {
                int k = i * n + j;
                if(k > 0)
                    add(new Button("" + k));
            }
        }
    }
}
```

Following is sample output generated by the **GridLayoutDemo** applet:



TIP You might try using this example as the starting point for a 15-square puzzle.

CardLayout

The **CardLayout** class is unique among the other layout managers in that it stores several different layouts. Each layout can be thought of as being on a separate index card in a deck that can be shuffled so that any card is on top at a given time. This can be useful for user interfaces with optional components that can be dynamically enabled and disabled upon user input. You can prepare the other layouts and have them hidden, ready to be activated when needed.

CardLayout provides these two constructors:

```
CardLayout()
CardLayout(int horz, int vert)
```

The first form creates a default card layout. The second form allows you to specify the horizontal and vertical space left between components in *horz* and *vert*, respectively.

Use of a card layout requires a bit more work than the other layouts. The cards are typically held in an object of type **Panel**. This panel must have **CardLayout** selected as its layout manager. The cards that form the deck are also typically objects of type **Panel**. Thus, you must create a panel that contains the deck and a panel for each card in the deck. Next, you add to the appropriate panel the components that form each card. You then add these panels to the panel for which **CardLayout** is the layout manager. Finally, you add this panel to the window. Once these steps are complete, you must provide some way for the user to select between cards. One common approach is to include one push button for each card in the deck.

When card panels are added to a panel, they are usually given a name. Thus, most of the time, you will use this form of **add()** when adding cards to a panel:

```
void add(Component panelRef, Object name)
```

Here, *name* is a string that specifies the name of the card whose panel is specified by *panelRef*.

After you have created a deck, your program activates a card by calling one of the following methods defined by **CardLayout**:

```
void first(Container deck)
void last(Container deck)
void next(Container deck)
void previous(Container deck)
void show(Container deck, String cardName)
```

Here, *deck* is a reference to the container (usually a panel) that holds the cards, and *cardName* is the name of a card. Calling **first()** causes the first card in the deck to be shown. To show the last card, call **last()**. To show the next card, call **next()**. To show the previous card, call **previous()**. Both **next()** and **previous()** automatically cycle back to the top or bottom of the deck, respectively. The **show()** method displays the card whose name is passed in *cardName*.

The following example creates a two-level card deck that allows the user to select an operating system. Windows-based operating systems are displayed in one card. Mac OS and Solaris are displayed in the other card.

```
// Demonstrate CardLayout.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="CardLayoutDemo" width=300 height=100>
</applet>
*/
public class CardLayoutDemo extends Applet
implements ActionListener, MouseListener {
```

```
Checkbox windowsXP, windows7, windows8, android, solaris, mac;
Panel osCards;
CardLayout cardLO;
Button Win, Other;

public void init() {
    Win = new Button("Windows");
    Other = new Button("Other");
    add(Win);
    add(Other);

    cardLO = new CardLayout();
    osCards = new Panel();
    osCards.setLayout(cardLO); // set panel layout to card layout

    windowsXP = new Checkbox("Windows XP", null, true);
    windows7 = new Checkbox("Windows 7", null, false);
    windows8 = new Checkbox("Windows 8", null, false);
    android = new Checkbox("Android");
    solaris = new Checkbox("Solaris");
    mac = new Checkbox("Mac OS");

    // add Windows check boxes to a panel
    Panel winPan = new Panel();
    winPan.add(windowsXP);
    winPan.add(windows7);
    winPan.add(windows8);

    // Add other OS check boxes to a panel
    Panel otherPan = new Panel();
    otherPan.add(android);
    otherPan.add(solaris);
    otherPan.add(mac);

    // add panels to card deck panel
    osCards.add(winPan, "Windows");
    osCards.add(otherPan, "Other");

    // add cards to main applet panel
    add(osCards);

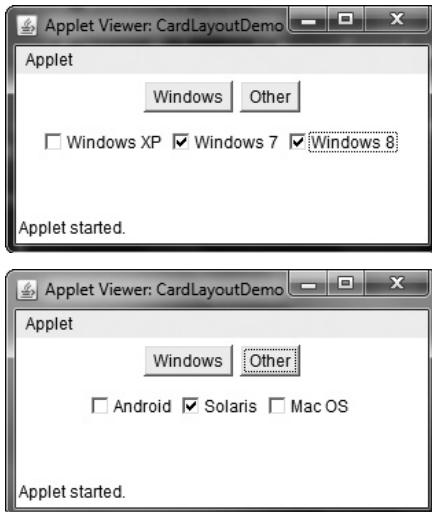
    // register to receive action events
    Win.addActionListener(this);
    Other.addActionListener(this);

    // register mouse events
    addMouseListener(this);
}

// Cycle through panels.
public void mousePressed(MouseEvent me) {
    cardLO.next(osCards);
}
```

```
// Provide empty implementations for the other MouseListener methods.  
public void mouseClicked(MouseEvent me) {  
}  
public void mouseEntered(MouseEvent me) {  
}  
public void mouseExited(MouseEvent me) {  
}  
public void mouseReleased(MouseEvent me) {  
}  
  
public void actionPerformed(ActionEvent ae) {  
    if(ae.getSource() == Win) {  
  
        cardLO.show(osCards, "Windows");  
    }  
    else {  
        cardLO.show(osCards, "Other");  
    }  
}
```

Here is sample output generated by the **CardLayoutDemo** applet. Each card is activated by pushing its button. You can also cycle through the cards by clicking the mouse.



GridBagLayout

Although the preceding layouts are perfectly acceptable for many uses, some situations will require that you take a bit more control over how the components are arranged. A good way to do this is to use a grid bag layout, which is specified by the **GridBagLayout** class. What makes the grid bag useful is that you can specify the relative placement of components by specifying their positions within cells inside a grid. The key to the grid bag is that each component can be a different size, and each row in the grid can have a different number

of columns. This is why the layout is called a *grid bag*. It's a collection of small grids joined together.

The location and size of each component in a grid bag are determined by a set of constraints linked to it. The constraints are contained in an object of type **GridBagConstraints**. Constraints include the height and width of a cell, and the placement of a component, its alignment, and its anchor point within the cell.

The general procedure for using a grid bag is to first create a new **GridBagLayout** object and to make it the current layout manager. Then, set the constraints that apply to each component that will be added to the grid bag. Finally, add the components to the layout manager. Although **GridBagLayout** is a bit more complicated than the other layout managers, it is still quite easy to use once you understand how it works.

GridBagLayout defines only one constructor, which is shown here:

```
GridBagLayout( )
```

GridBagLayout defines several methods, of which many are protected and not for general use. There is one method, however, that you must use: **setConstraints()**. It is shown here:

```
void setConstraints(Component comp, GridBagConstraints cons)
```

Here, *comp* is the component for which the constraints specified by *cons* apply. This method sets the constraints that apply to each component in the grid bag.

The key to successfully using **GridBagLayout** is the proper setting of the constraints, which are stored in a **GridBagConstraints** object. **GridBagConstraints** defines several fields that you can set to govern the size, placement, and spacing of a component. These are shown in Table 26-1. Several are described in greater detail in the following discussion.

Field	Purpose
int anchor	Specifies the location of a component within a cell. The default is GridBagConstraints.CENTER .
int fill	Specifies how a component is resized if the component is smaller than its cell. Valid values are GridBagConstraints.NONE (the default), GridBagConstraints.HORIZONTAL , GridBagConstraints.VERTICAL , GridBagConstraints.BOTH .
int gridheight	Specifies the height of component in terms of cells. The default is 1.
int gridwidth	Specifies the width of component in terms of cells. The default is 1.
int gridx	Specifies the X coordinate of the cell to which the component will be added. The default value is GridBagConstraints.RELATIVE .
int gridy	Specifies the Y coordinate of the cell to which the component will be added. The default value is GridBagConstraints.RELATIVE .
Insets insets	Specifies the insets. Default insets are all zero.
int ipadx	Specifies extra horizontal space that surrounds a component within a cell. The default is 0.
int ipady	Specifies extra vertical space that surrounds a component within a cell. The default is 0.

Table 26-1 Constraint Fields Defined by **GridBagConstraints**

Field	Purpose
double weightx	Specifies a weight value that determines the horizontal spacing between cells and the edges of the container that holds them. The default value is 0.0. The greater the weight, the more space that is allocated. If all values are 0.0, extra space is distributed evenly between the edges of the window.
double weighty	Specifies a weight value that determines the vertical spacing between cells and the edges of the container that holds them. The default value is 0.0. The greater the weight, the more space that is allocated. If all values are 0.0, extra space is distributed evenly between the edges of the window.

Table 26-1 Constraint Fields Defined by **GridBagConstraints** (*continued*)

GridBagConstraints also defines several static fields that contain standard constraint values, such as **GridBagConstraints.CENTER** and **GridBagConstraints.VERTICAL**.

When a component is smaller than its cell, you can use the **anchor** field to specify where within the cell the component's top-left corner will be located. There are three types of values that you can give to **anchor**. The first are absolute:

GridBagConstraints.CENTER	GridBagConstraints.SOUTH
GridBagConstraints.EAST	GridBagConstraints.SOUTHEAST
GridBagConstraints.NORTH	GridBagConstraints.SOUTHWEST
GridBagConstraints.NORTHEAST	GridBagConstraints.WEST
GridBagConstraints.NORTHWEST	

As their names imply, these values cause the component to be placed at the specific locations.

The second type of values that can be given to **anchor** is relative, which means the values are relative to the container's orientation, which might differ for non-Western languages. The relative values are shown here:

GridBagConstraints.FIRST_LINE_END	GridBagConstraints.LINE_END
GridBagConstraints.FIRST_LINE_START	GridBagConstraints.LINE_START
GridBagConstraints.LAST_LINE_END	GridBagConstraints.PAGE_END
GridBagConstraints.LAST_LINE_START	GridBagConstraints.PAGE_START

Their names describe the placement.

The third type of values that can be given to **anchor** allows you to position components relative to the baseline of the row. These values are shown here:

GridBagConstraints.BASELINE	GridBagConstraints.BASELINE.LEADING
GridBagConstraints.BASELINE.TRAILING	GridBagConstraints.ABOVE_BASELINE
GridBagConstraints.ABOVE_BASELINE.LEADING	GridBagConstraints.ABOVE_BASELINE.TRAILING
GridBagConstraints.BELOW_BASELINE	GridBagConstraints.BELOW_BASELINE.LEADING
GridBagConstraints.BELOW_BASELINE.TRAILING	

The horizontal position can be either centered, against the leading edge (LEADING), or against the trailing edge (TRAILING).

The **weightx** and **weighty** fields are both quite important and quite confusing at first glance. In general, their values determine how much of the extra space within a container is allocated to each row and column. By default, both these values are zero. When all values within a row or a column are zero, extra space is distributed evenly between the edges of the window. By increasing the weight, you increase that row or column's allocation of space proportional to the other rows or columns. The best way to understand how these values work is to experiment with them a bit.

The **gridwidth** variable lets you specify the width of a cell in terms of cell units. The default is 1. To specify that a component use the remaining space in a row, use **GridBagConstraints.REMAINDER**. To specify that a component use the next-to-last cell in a row, use **GridBagConstraints.RELATIVE**. The **gridheight** constraint works the same way, but in the vertical direction.

You can specify a padding value that will be used to increase the minimum size of a cell. To pad horizontally, assign a value to **ipadx**. To pad vertically, assign a value to **ipady**.

Here is an example that uses **GridBagLayout** to demonstrate several of the points just discussed:

```
// Use GridBagLayout.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="GridBagDemo" width=250 height=200>
</applet>
*/
public class GridBagDemo extends Applet
    implements ItemListener {

    String msg = "";
    Checkbox windows, android, solaris, mac;

    public void init() {
        GridBagLayout qbag = new GridBagLayout();
        GridBagConstraints gbc = new GridBagConstraints();
        setLayout(qbag);

        // Define check boxes.
        windows = new Checkbox("Windows ", null, true);
        android = new Checkbox("Android");
        solaris = new Checkbox("Solaris");
        mac = new Checkbox("Mac OS");

        // Define the grid bag.

        // Use default row weight of 0 for first row.
        gbc.weightx = 1.0; // use a column weight of 1
```

```
gbc.ipadx = 200; // pad by 200 units
gbc.insets = new Insets(4, 4, 0, 0); // inset slightly from top left

gbc.anchor = GridBagConstraints.NORTHEAST;

gbc.gridwidth = GridBagConstraints.RELATIVE;
gbag.setConstraints(windows, gbc);

gbc.gridwidth = GridBagConstraints.REMAINDER;
gbag.setConstraints(android, gbc);

// Give second row a weight of 1.
gbc.weighty = 1.0;

gbc.gridwidth = GridBagConstraints.RELATIVE;
gbag.setConstraints(solaris, gbc);

gbc.gridwidth = GridBagConstraints.REMAINDER;
gbag.setConstraints(mac, gbc);

// Add the components.
add(windows);
add(android);
add(solaris);
add(mac);

// Register to receive item events.
windows.addItemListener(this);
android.addItemListener(this);
solaris.addItemListener(this);
mac.addItemListener(this);
}

// Repaint when status of a check box changes.
public void itemStateChanged(ItemEvent ie) {
    repaint();
}

// Display current state of the check boxes.
public void paint(Graphics g) {
    msg = "Current state: ";
    g.drawString(msg, 6, 80);
    msg = " Windows: " + windows.getState();
    g.drawString(msg, 6, 100);
    msg = " Android: " + android.getState();
    g.drawString(msg, 6, 120);
    msg = " Solaris: " + solaris.getState();
    g.drawString(msg, 6, 140);
    msg = " Mac: " + mac.getState();
    g.drawString(msg, 6, 160);
}
}
```

Sample output produced by the program is shown here.



In this layout, the operating system check boxes are positioned in a 2×2 grid. Each cell has a horizontal padding of 200. Each component is inset slightly (by 4 units) from the top left. The column weight is set to 1, which causes any extra horizontal space to be distributed evenly between the columns. The first row uses a default weight of 0; the second has a weight of 1. This means that any extra vertical space is added to the second row.

GridBagLayout is a powerful layout manager. It is worth taking some time to experiment with and explore. Once you understand what the various settings do, you can use **GridBagLayout** to position components with a high degree of precision.

Menu Bars and Menus

A top-level window can have a menu bar associated with it. A menu bar displays a list of top-level menu choices. Each choice is associated with a drop-down menu. This concept is implemented in the AWT by the following classes: **MenuBar**, **Menu**, and **MenuItem**. In general, a menu bar contains one or more **Menu** objects. Each **Menu** object contains a list of **MenuItem** objects. Each **MenuItem** object represents something that can be selected by the user. Since **Menu** is a subclass of **MenuItem**, a hierarchy of nested submenus can be created. It is also possible to include checkable menu items. These are menu options of type **CheckboxMenuItem** and will have a check mark next to them when they are selected.

To create a menu bar, first create an instance of **MenuBar**. This class defines only the default constructor. Next, create instances of **Menu** that will define the selections displayed on the bar. Following are the constructors for **Menu**:

```
Menu( ) throws HeadlessException
Menu(String optionName) throws HeadlessException
Menu(String optionName, boolean removable) throws HeadlessException
```

Here, *optionName* specifies the name of the menu selection. If *removable* is **true**, the menu can be removed and allowed to float free. Otherwise, it will remain attached to the menu bar. (Removable menus are implementation-dependent.) The first form creates an empty menu.

Individual menu items are of type **MenuItem**. It defines these constructors:

```
MenuItem( ) throws HeadlessException
MenuItem(String itemName) throws HeadlessException
MenuItem(String itemName, MenuShortcut keyAccel) throws HeadlessException
```

Here, *itemName* is the name shown in the menu, and *keyAccel* is the menu shortcut for this item.

You can disable or enable a menu item by using the **setEnabled()** method. Its form is shown here:

```
void setEnabled(boolean enabledFlag)
```

If the argument *enabledFlag* is **true**, the menu item is enabled. If **false**, the menu item is disabled.

You can determine an item's status by calling **isEnabled()**. This method is shown here:

```
boolean isEnabled()
```

isEnabled() returns **true** if the menu item on which it is called is enabled. Otherwise, it returns **false**.

You can change the name of a menu item by calling **setLabel()**. You can retrieve the current name by using **getLabel()**. These methods are as follows:

```
void setLabel(String newName)
```

```
String getLabel()
```

Here, *newName* becomes the new name of the invoking menu item. **getLabel()** returns the current name.

You can create a checkable menu item by using a subclass of **MenuItem** called **CheckboxMenuItem**. It has these constructors:

```
CheckboxMenuItem() throws HeadlessException
```

```
CheckboxMenuItem(String itemName) throws HeadlessException
```

```
CheckboxMenuItem(String itemName, boolean on) throws HeadlessException
```

Here, *itemName* is the name shown in the menu. Checkable items operate as toggles. Each time one is selected, its state changes. In the first two forms, the checkable entry is unchecked. In the third form, if *on* is **true**, the checkable entry is initially checked. Otherwise, it is cleared.

You can obtain the status of a checkable item by calling **getState()**. You can set it to a known state by using **setState()**. These methods are shown here:

```
boolean getState()
```

```
void setState(boolean checked)
```

If the item is checked, **getState()** returns **true**. Otherwise, it returns **false**. To check an item, pass **true** to **setState()**. To clear an item, pass **false**.

Once you have created a menu item, you must add the item to a **Menu** object by using **add()**, which has the following general form:

```
MenuItem add(MenuItem item)
```

Here, *item* is the item being added. Items are added to a menu in the order in which the calls to **add()** take place. The *item* is returned.

Once you have added all items to a **Menu** object, you can add that object to the menu bar by using this version of **add()** defined by **MenuBar**:

```
Menu add(Menu menu)
```

Here, *menu* is the menu being added. The *menu* is returned.

Menus generate events only when an item of type **MenuItem** or **CheckboxMenuItem** is selected. They do not generate events when a menu bar is accessed to display a drop-down menu, for example. Each time a menu item is selected, an **ActionEvent** object is generated. By default, the action command string is the name of the menu item. However, you can specify a different action command string by calling **setActionCommand()** on the menu item. Each time a check box menu item is checked or unchecked, an **ItemEvent** object is generated. Thus, you must implement the **ActionListener** and/or **ItemListener** interfaces in order to handle these menu events.

The **getItem()** method of **ItemEvent** returns a reference to the item that generated this event. The general form of this method is shown here:

```
Object getItem( )
```

Following is an example that adds a series of nested menus to a pop-up window. The item selected is displayed in the window. The state of the two check box menu items is also displayed.

```
// Illustrate menus.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="MenuDemo" width=250 height=250>
</applet>
*/
// Create a subclass of Frame.
class MenuFrame extends Frame {
    String msg = "";
    CheckboxMenuItem debug, test;

    MenuFrame(String title) {
        super(title);

        // create menu bar and add it to frame
        MenuBar mbar = new MenuBar();
        setMenuBar(mbar);

        // create the menu items
        Menu file = new Menu("File");
        MenuItem item1, item2, item3, item4, item5;
        file.add(item1 = new MenuItem("New..."));
        file.add(item2 = new MenuItem("Open..."));
        file.add(item3 = new MenuItem("Close"));
        file.add(item4 = new MenuItem("-"));
        file.add(item5 = new MenuItem("Quit..."));
        mbar.add(file);

        Menu edit = new Menu("Edit");
        MenuItem item6, item7, item8, item9;
        edit.add(item6 = new MenuItem("Cut"));
        edit.add(item7 = new MenuItem("Copy"));

        // Create a submenu
        Menu sub = new Menu("Submenu");
        sub.add(item1 = new MenuItem("Sub1"));
        sub.add(item2 = new MenuItem("Sub2"));
        sub.add(item3 = new MenuItem("Sub3"));
        edit.add(sub);
    }
}
```

```
edit.add(item8 = new MenuItem("Paste"));
edit.add(item9 = new MenuItem("-"));

Menu sub = new Menu("Special");
MenuItem item10, item11, item12;
sub.add(item10 = new MenuItem("First"));
sub.add(item11 = new MenuItem("Second"));
sub.add(item12 = new MenuItem("Third"));
edit.add(sub);

// these are checkable menu items
debug = new CheckboxMenuItem("Debug");
edit.add(debug);
test = new CheckboxMenuItem("Testing");
edit.add(test);

mbar.add(edit);

// create an object to handle action and item events
MyMenuHandler handler = new MyMenuHandler(this);
// register it to receive those events
item1.addActionListener(handler);
item2.addActionListener(handler);
item3.addActionListener(handler);
item4.addActionListener(handler);
item5.addActionListener(handler);
item6.addActionListener(handler);
item7.addActionListener(handler);
item8.addActionListener(handler);
item9.addActionListener(handler);
item10.addActionListener(handler);
item11.addActionListener(handler);
item12.addActionListener(handler);
debug.addItemListener(handler);
test.addItemListener(handler);

// create an object to handle window events
MyWindowAdapter adapter = new MyWindowAdapter(this);
// register it to receive those events
addWindowListener(adapter);
}

public void paint(Graphics g) {
    g.drawString(msg, 10, 200);

    if(debug.getState())
        g.drawString("Debug is on.", 10, 220);
    else
        g.drawString("Debug is off.", 10, 220);

    if(test.getState())
        g.drawString("Testing is on.", 10, 240);
    else
```

```
        g.drawString("Testing is off.", 10, 240);
    }
}

class MyWindowAdapter extends WindowAdapter {
    MenuFrame menuFrame;

    public MyWindowAdapter(MenuFrame menuFrame) {
        this.menuFrame = menuFrame;
    }

    public void windowClosing(WindowEvent we) {
        menuFrame.setVisible(false);
    }
}

class MyMenuHandler implements ActionListener, ItemListener {
    MenuFrame menuFrame;

    public MyMenuHandler(MenuFrame menuFrame) {
        this.menuFrame = menuFrame;
    }

    // Handle action events.
    public void actionPerformed(ActionEvent ae) {
        String msg = "You selected ";
        String arg = ae.getActionCommand();
        if(arg.equals("New..."))
            msg += "New.";
        else if(arg.equals("Open..."))
            msg += "Open.";
        else if(arg.equals("Close"))
            msg += "Close.";
        else if(arg.equals("Quit..."))
            msg += "Quit.";
        else if(arg.equals("Edit"))
            msg += "Edit.";
        else if(arg.equals("Cut"))
            msg += "Cut.";
        else if(arg.equals("Copy"))
            msg += "Copy.";
        else if(arg.equals("Paste"))
            msg += "Paste.";
        else if(arg.equals("First"))
            msg += "First.";
        else if(arg.equals("Second"))
            msg += "Second.";
        else if(arg.equals("Third"))
            msg += "Third.";
        else if(arg.equals("Debug"))
            msg += "Debug.";
        else if(arg.equals("Testing"))
            msg += "Testing.";
    }
}
```

```
menuFrame.msg = msg;
menuFrame.repaint();
}

// Handle item events.
public void itemStateChanged(ItemEvent ie) {
    menuFrame.repaint();
}

// Create frame window.
public class MenuDemo extends Applet {
    Frame f;

    public void init() {
        f = new MenuFrame("Menu Demo");
        int width = Integer.parseInt(getParameter("width"));
        int height = Integer.parseInt(getParameter("height"));

        setSize(new Dimension(width, height));

        f.setSize(width, height);
        f.setVisible(true);
    }

    public void start() {
        f.setVisible(true);
    }

    public void stop() {
        f.setVisible(false);
    }
}
```

Sample output from the **MenuDemo** applet is shown in Figure 26-8.

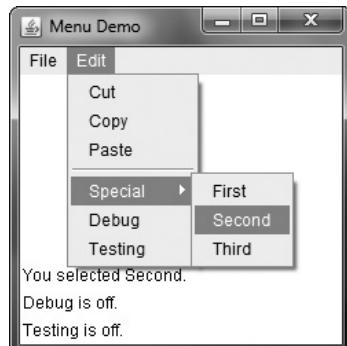


Figure 26-8 Sample output from the **MenuDemo** applet

There is one other menu-related class that you might find interesting: **PopupMenu**. It works just like **Menu**, but produces a menu that can be displayed at a specific location. **PopupMenu** provides a flexible, useful alternative for some types of menuing situations.

Dialog Boxes

Often, you will want to use a *dialog box* to hold a set of related controls. Dialog boxes are primarily used to obtain user input and are often child windows of a top-level window. Dialog boxes don't have menu bars, but in other respects, they function like frame windows. (You can add controls to them, for example, in the same way that you add controls to a frame window.) Dialog boxes may be modal or modeless. When a *modal* dialog box is active, all input is directed to it until it is closed. This means that you cannot access other parts of your program until you have closed the dialog box. When a *modeless* dialog box is active, input focus can be directed to another window in your program. Thus, other parts of your program remain active and accessible. In the AWT, dialog boxes are of type **Dialog**. Two commonly used constructors are shown here:

```
Dialog(Frame parentWindow, boolean mode)
Dialog(Frame parentWindow, String title, boolean mode)
```

Here, *parentWindow* is the owner of the dialog box. If *mode* is **true**, the dialog box is modal. Otherwise, it is modeless. The title of the dialog box can be passed in *title*. Generally, you will subclass **Dialog**, adding the functionality required by your application.

Following is a modified version of the preceding menu program that displays a modeless dialog box when the New option is chosen. Notice that when the dialog box is closed, **dispose()** is called. This method is defined by **Window**, and it frees all system resources associated with the dialog box window.

```
// Demonstrate Dialog box.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="DialogDemo" width=250 height=250>
</applet>
*/
// Create a subclass of Dialog.
class SampleDialog extends Dialog implements ActionListener {
    SampleDialog(Frame parent, String title) {
        super(parent, title, false);
        setLayout(new FlowLayout());
        setSize(300, 200);

        add(new Label("Press this button:"));
        Button b;
        add(b = new Button("Cancel"));
        b.addActionListener(this);
    }
}
```

```
public void actionPerformed(ActionEvent ae) {
    dispose();
}

public void paint(Graphics g) {
    g.drawString("This is in the dialog box", 10, 70);
}
}

// Create a subclass of Frame.
class MenuFrame extends Frame {
    String msg = "";
    CheckboxMenuItem debug, test;

    MenuFrame(String title) {
        super(title);

        // create menu bar and add it to frame
        MenuBar mbar = new MenuBar();
        setMenuBar(mbar);

        // create the menu items
        Menu file = new Menu("File");
        MenuItem item1, item2, item3, item4;
        file.add(item1 = new MenuItem("New..."));
        file.add(item2 = new MenuItem("Open..."));
        file.add(item3 = new MenuItem("Close"));
        file.add(new MenuItem("-"));
        file.add(item4 = new MenuItem("Quit..."));
        mbar.add(file);

        Menu edit = new Menu("Edit");
        MenuItem item5, item6, item7;
        edit.add(item5 = new MenuItem("Cut"));
        edit.add(item6 = new MenuItem("Copy"));
        edit.add(item7 = new MenuItem("Paste"));
        edit.add(new MenuItem("-"));

        Menu sub = new Menu("Special", true);
        MenuItem item8, item9, item10;
        sub.add(item8 = new MenuItem("First"));
        sub.add(item9 = new MenuItem("Second"));
        sub.add(item10 = new MenuItem("Third"));
        edit.add(sub);

        // these are checkable menu items
        debug = new CheckboxMenuItem("Debug");
        edit.add(debug);
        test = new CheckboxMenuItem("Testing");
        edit.add(test);

        mbar.add(edit);
    }
}
```

```
// create an object to handle action and item events
MyMenuHandler handler = new MyMenuHandler(this);
// register it to receive those events
item1.addActionListener(handler);
item2.addActionListener(handler);
item3.addActionListener(handler);
item4.addActionListener(handler);
item5.addActionListener(handler);
item6.addActionListener(handler);
item7.addActionListener(handler);
item8.addActionListener(handler);
item9.addActionListener(handler);
item10.addActionListener(handler);
debug.addItemListener(handler);
test.addItemListener(handler);

// create an object to handle window events
MyWindowAdapter adapter = new MyWindowAdapter(this);

// register it to receive those events
addWindowListener(adapter);
}

public void paint(Graphics g) {
    g.drawString(msg, 10, 200);

    if(debug.getState())
        g.drawString("Debug is on.", 10, 220);
    else
        g.drawString("Debug is off.", 10, 220);

    if(test.getState())
        g.drawString("Testing is on.", 10, 240);
    else
        g.drawString("Testing is off.", 10, 240);
}
}

class MyWindowAdapter extends WindowAdapter {
    MenuFrame menuFrame;

    public MyWindowAdapter(MenuFrame menuFrame) {
        this.menuFrame = menuFrame;
    }

    public void windowClosing(WindowEvent we) {
        menuFrame.dispose();
    }
}

class MyMenuHandler implements ActionListener, ItemListener {
    MenuFrame menuFrame;
```

```
public MyMenuHandler(MenuFrame menuFrame) {  
    this.menuFrame = menuFrame;  
}  
  
// Handle action events.  
public void actionPerformed(ActionEvent ae) {  
    String msg = "You selected ";  
    String arg = ae.getActionCommand();  
    // Activate a dialog box when New is selected.  
    if(arg.equals("New...")) {  
        msg += "New.";  
        SampleDialog d = new  
            SampleDialog(menuFrame, "New Dialog Box");  
        d.setVisible(true);  
    }  
    // Try defining other dialog boxes for these options.  
    else if(arg.equals("Open..."))  
        msg += "Open.";  
    else if(arg.equals("Close"))  
        msg += "Close.";  
    else if(arg.equals("Quit..."))  
        msg += "Quit.";  
    else if(arg.equals("Edit"))  
        msg += "Edit.";  
    else if(arg.equals("Cut"))  
        msg += "Cut.";  
    else if(arg.equals("Copy"))  
        msg += "Copy.";  
    else if(arg.equals("Paste"))  
        msg += "Paste.";  
    else if(arg.equals("First"))  
        msg += "First.";  
    else if(arg.equals("Second"))  
        msg += "Second.";  
    else if(arg.equals("Third"))  
        msg += "Third.";  
    else if(arg.equals("Debug"))  
        msg += "Debug.";  
    else if(arg.equals("Testing"))  
        msg += "Testing.";  
    menuFrame.msg = msg;  
    menuFrame.repaint();  
}  
  
public void itemStateChanged(ItemEvent ie) {  
    menuFrame.repaint();  
}  
}  
  
// Create frame window.  
public class DialogDemo extends Applet {  
    Frame f;
```

```
public void init() {
    f = new MenuFrame("Menu Demo");
    int width = Integer.parseInt(getParameter("width"));
    int height = Integer.parseInt(getParameter("height"));

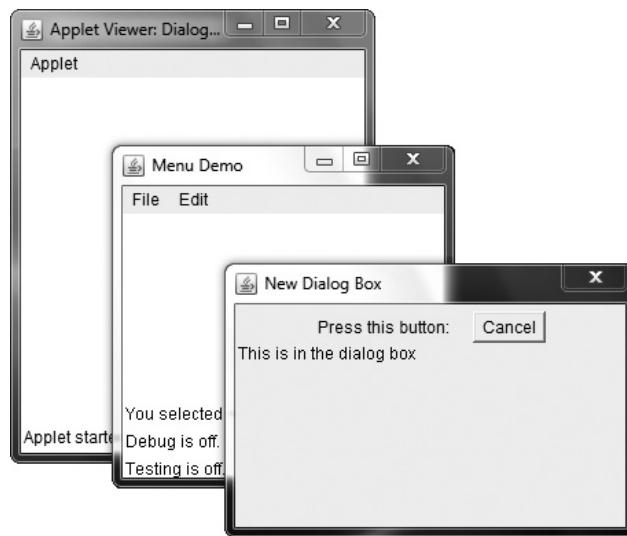
    setSize(width, height);

    f.setSize(width, height);
    f.setVisible(true);
}

public void start() {
    f.setVisible(true);
}

public void stop() {
    f.setVisible(false);
}
}
```

Here is sample output from the **DialogDemo** applet:



TIP On your own, try defining dialog boxes for the other options presented by the menus.

FileDialog

Java provides a built-in dialog box that lets the user specify a file. To create a file dialog box, instantiate an object of type **FileDialog**. This causes a file dialog box to be displayed.

Usually, this is the standard file dialog box provided by the operating system. Here are three **FileDialog** constructors:

```
FileDialog(Frame parent)
FileDialog(Frame parent, String boxName)
FileDialog(Frame parent, String boxName, int how)
```

Here, *parent* is the owner of the dialog box. The *boxName* parameter specifies the name displayed in the box's title bar. If *boxName* is omitted, the title of the dialog box is empty. If *how* is **FileDialog.LOAD**, then the box is selecting a file for reading. If *how* is **FileDialog.SAVE**, the box is selecting a file for writing. If *how* is omitted, the box is selecting a file for reading.

FileDialog provides methods that allow you to determine the name of the file and its path as selected by the user. Here are two examples:

```
String getDirectory()
String getFile()
```

These methods return the directory and the filename, respectively.

The following program activates the standard file dialog box:

```
/* Demonstrate File Dialog box.

    This is an application, not an applet.
*/
import java.awt.*;
import java.awt.event.*;

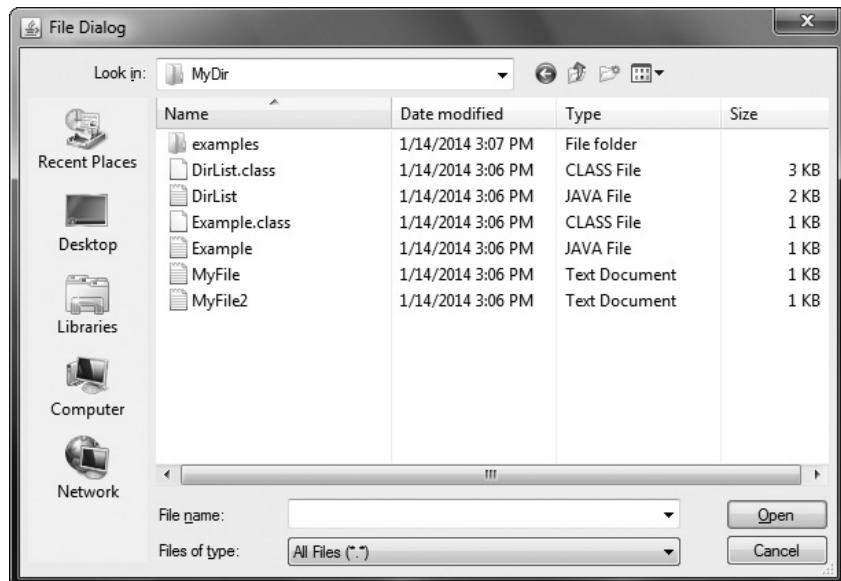
// Create a subclass of Frame.
class SampleFrame extends Frame {
    SampleFrame(String title) {
        super(title);

        // remove the window when closed
        addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent we) {
                System.exit(0);
            }
        });
    }

    // Demonstrate FileDialog.
    class FileDialogDemo {
        public static void main(String args[]) {
            // create a frame that owns the dialog
            Frame f = new SampleFrame("File Dialog Demo");
            f.setVisible(true);
            f.setSize(100, 100);

            FileDialog fd = new FileDialog(f, "File Dialog");
            fd.setVisible(true);
        }
    }
}
```

The output generated by this program is shown here. (The precise configuration of the dialog box may vary.)



One last point: Beginning with JDK 7, you can use **FileDialog** to select a list of files. This functionality is supported by the **setMultipleMode()**, **isMultipleMode()**, and **getFiles()** methods.

A Word About Overriding **paint()**

Before concluding our examination of AWT controls, a short word about overriding **paint()** is in order. Although not relevant to the simple AWT examples shown in this book, when overriding **paint()**, there are times when it is necessary to call the superclass implementation of **paint()**. Therefore, for some programs, you will need to use this **paint()** skeleton:

```
public void paint(Graphics g) {  
    // code to repaint this window  
  
    // Call superclass paint()  
    super.paint(g);  
}
```

In Java, there are two general types of components: heavyweight and lightweight. A heavyweight component has its own native window, which is called its *peer*. A lightweight component is implemented completely in Java code and uses the window provided by an ancestor. The AWT controls described and used in this chapter are all heavyweight. However, if a container holds any lightweight components (that is, has lightweight child components), your override of `paint()` for that container must call `super.paint()`. By calling `super.paint()`, you ensure that any lightweight child components, such as lightweight controls, get properly painted. If you are unsure of a child component's type, you can call `isLightweight()`, defined by **Component**, to find out. It returns **true** if the component is lightweight, and **false** otherwise.

This page has been intentionally left blank

CHAPTER

27

Images

This chapter examines the **Image** class and the **java.awt.image** package. Together, they provide support for *imaging* (the display and manipulation of graphical images). An *image* is simply a rectangular graphical object. Images are a key component of web design. In fact, the inclusion of the tag in the Mosaic browser at NCSA (National Center for Supercomputer Applications) is what caused the Web to begin to grow explosively in 1993. This tag was used to include an image *inline* with the flow of hypertext. Java expands upon this basic concept, allowing images to be managed under program control. Because of its importance, Java provides extensive support for imaging.

Images are objects of the **Image** class, which is part of the **java.awt** package. Images are manipulated using the classes found in the **java.awt.image** package. There are a large number of imaging classes and interfaces defined by **java.awt.image**, and it is not possible to examine them all. Instead, we will focus on those that form the foundation of imaging. Here are the **java.awt.image** classes discussed in this chapter:

CropImageFilter	MemoryImageSource
FilteredImageSource	PixelGrabber
ImageFilter	RGBImageFilter

These are the interfaces that we will use:

ImageConsumer	ImageObserver	ImageProducer
---------------	---------------	---------------

Also examined is the **MediaTracker** class, which is part of **java.awt**.

File Formats

Originally, web images could only be in GIF format. The GIF image format was created by CompuServe in 1987 to make it possible for images to be viewed while online, so it was well suited to the Internet. GIF images can have only up to 256 colors each. This limitation

caused the major browser vendors to add support for JPEG images in 1995. The JPEG format was created by a group of photographic experts to store full-color-spectrum, continuous-tone images. These images, when properly created, can be of much higher fidelity as well as more highly compressed than a GIF encoding of the same source image. Another file format is PNG. It too is an alternative to GIF. In almost all cases, you will never care or notice which format is being used in your programs. The Java image classes abstract the differences behind a clean interface.

Image Fundamentals: Creating, Loading, and Displaying

There are three common operations that occur when you work with images: creating an image, loading an image, and displaying an image. In Java, the **Image** class is used to refer to images in memory and to images that must be loaded from external sources. Thus, Java provides ways for you to create a new image object and ways to load one. It also provides a means by which an image can be displayed. Let's look at each.

Creating an Image Object

You might expect that you create a memory image using something like the following:

```
Image test = new Image(200, 100); // Error -- won't work
```

Not so. Because images must eventually be painted on a window to be seen, the **Image** class doesn't have enough information about its environment to create the proper data format for the screen. Therefore, the **Component** class in **java.awt** has a factory method called **createImage()** that is used to create **Image** objects. (Remember that all of the AWT components are subclasses of **Component**, so all support this method.)

The **createImage()** method has the following two forms:

```
Image createImage(ImageProducer imgProd)  
Image createImage(int width, int height)
```

The first form returns an image produced by *imgProd*, which is an object of a class that implements the **ImageProducer** interface. (We will look at image producers later.) The second form returns a blank (that is, empty) image that has the specified width and height. Here is an example:

```
Canvas c = new Canvas();  
Image test = c.createImage(200, 100);
```

This creates an instance of **Canvas** and then calls the **createImage()** method to actually make an **Image** object. At this point, the image is blank. Later, you will see how to write data to it.

Loading an Image

The other way to obtain an image is to load one. One way to do this is to use the **getImage()** method defined by the **Applet** class. It has the following forms:

```
Image getImage(URL url)  
Image getImage(URL url, String imageName)
```

The first version returns an **Image** object that encapsulates the image found at the location specified by *url*. The second version returns an **Image** object that encapsulates the image found at the location specified by *url* and having the name specified by *imageName*.

Displaying an Image

Once you have an image, you can display it by using **drawImage()**, which is a member of the **Graphics** class. It has several forms. The one we will be using is shown here:

```
boolean drawImage(Image imgObj, int left, int top, ImageObserver imgOb)
```

This displays the image passed in *imgObj* with its upper-left corner specified by *left* and *top*. *imgOb* is a reference to a class that implements the **ImageObserver** interface. This interface is implemented by all AWT (and Swing) components. An *image observer* is an object that can monitor an image while it loads. **ImageObserver** is described in the next section.

With **getImage()** and **drawImage()**, it is actually quite easy to load and display an image. Here is a sample applet that loads and displays a single image. The file **Lilies.jpg** is loaded, but you can substitute any GIF, JPG, or PNG file you like (just make sure it is available in the same directory with the HTML file that contains the applet).

```
/*
 * <applet code="SimpleImageLoad" width=400 height=345>
 *   <param name="img" value="Lilies.jpg">
 * </applet>
 */
import java.awt.*;
import java.applet.*;

public class SimpleImageLoad extends Applet
{
    Image img;

    public void init() {
        img = getImage(getDocumentBase(), getParameter("img"));
    }

    public void paint(Graphics g) {
        g.drawImage(img, 0, 0, this);
    }
}
```

In the **init()** method, the **img** variable is assigned to the image returned by **getImage()**. The **getImage()** method uses the string returned by **getParameter("img")** as the filename for the image. This image is loaded from a **URL** that is relative to the result of **getDocumentBase()**, which is the **URL** of the HTML page this applet tag was in. The filename returned by **getParameter("img")** comes from the applet tag **<param name="img" value="Lilies.jpg">**. This is the equivalent, if a little slower, of using the HTML tag ****. Figure 27-1 shows what it looks like when you run the program.

When this applet runs, it starts loading **img** in the **init()** method. Onscreen you can see the image as it loads from the network, because **Applet**'s implementation of the **ImageObserver** interface calls **paint()** every time more image data arrives.



Figure 27-1 Sample output from **SimpleImageLoad**

Seeing the image load is somewhat informative, but it might be better if you use the time it takes to load the image to do other things in parallel. That way, the fully formed image can simply appear on the screen in an instant, once it is fully loaded. You can use **ImageObserver**, described next, to monitor loading an image while you paint the screen with other information.

ImageObserver

ImageObserver is an interface used to receive notification as an image is being generated, and it defines only one method: **imageUpdate()**. Using an image observer allows you to perform other actions, such as show a progress indicator or an attract screen, as you are informed of the progress of the download. This kind of notification is very useful when an image is being loaded over a slow network.

The **imageUpdate()** method has this general form:

```
boolean imageUpdate(Image imgObj, int flags, int left, int top,  
                    int width, int height)
```

Here, *imgObj* is the image being loaded, and *flags* is an integer that communicates the status of the update report. The four integers *left*, *top*, *width*, and *height* represent a rectangle that contains different values depending on the values passed in *flags*. **imageUpdate()** should return **false** if it has completed loading, and **true** if there is more image to process.

The *flags* parameter contains one or more bit flags defined as static variables inside the **ImageObserver** interface. These flags and the information they provide are listed in Table 27-1.

Flag	Meaning
WIDTH	The <i>width</i> parameter is valid and contains the width of the image.
HEIGHT	The <i>height</i> parameter is valid and contains the height of the image.
PROPERTIES	The properties associated with the image can now be obtained using <code>imgObj.getProperty()</code> .
SOMEBITS	More pixels needed to draw the image have been received. The parameters <i>left</i> , <i>top</i> , <i>width</i> , and <i>height</i> define the rectangle containing the new pixels.
FRAMEBITS	A complete frame that is part of a multiframe image, which was previously drawn, has been received. This frame can be displayed. The <i>left</i> , <i>top</i> , <i>width</i> , and <i>height</i> parameters are not used.
ALLBITS	The image is now complete. The <i>left</i> , <i>top</i> , <i>width</i> , and <i>height</i> parameters are not used.
ERROR	An error has occurred to an image that was being tracked asynchronously. The image is incomplete and cannot be displayed. No further image information will be received. The ABORT flag will also be set to indicate that the image production was aborted.
ABORT	An image that was being tracked asynchronously was aborted before it was complete. However, if an error has not occurred, accessing any part of the image's data will restart the production of the image.

Table 27-1 Bit Flags of the `imageUpdate()` *flags* Parameter

The **Applet** class has an implementation of the `imageUpdate()` method for the **ImageObserver** interface that is used to repaint images as they are loaded. You can override this method in your class to change that behavior.

Here is a simple example of an `imageUpdate()` method:

```
public boolean imageUpdate(Image img, int flags,
                           int x, int y, int w, int h) {
    if ((flags & ALLBITS) == 0) {
        System.out.println("Still processing the image.");
        return true;
    } else {
        System.out.println("Done processing the image.");
        return false;
    }
}
```

Double Buffering

Not only are images useful for storing pictures, as we've just shown, but you can also use them as offscreen drawing surfaces. This allows you to render any image, including text and graphics, to an offscreen buffer that you can display at a later time. The advantage to doing this is that the image is seen only when it is complete. Drawing a complicated image could take several milliseconds or more, which can be seen by the user as flashing or flickering. This flashing is distracting and causes the user to perceive your rendering as slower than it actually is. Use of an offscreen image to reduce flicker is called *double buffering*, because the

screen is considered a buffer for pixels, and the offscreen image is the second buffer, where you can prepare pixels for display.

Earlier in this chapter, you saw how to create a blank **Image** object. Now you will see how to draw on that image rather than the screen. As you recall from earlier chapters, you need a **Graphics** object in order to use any of Java's rendering methods. Conveniently, the **Graphics** object that you can use to draw on an **Image** is available via the **getGraphics()** method. Here is a code fragment that creates a new image, obtains its graphics context, and fills the entire image with red pixels:

```
Canvas c = new Canvas();
Image test = c.createImage(200, 100);
Graphics gc = test.getGraphics();
gc.setColor(Color.red);
gc.fillRect(0, 0, 200, 100);
```

Once you have constructed and filled an offscreen image, it will still not be visible. To actually display the image, call **drawImage()**. Here is an example that draws a time-consuming image to demonstrate the difference that double buffering can make in perceived drawing time:

```
/*
 * <applet code=DoubleBuffer width=250 height=250>
 * </applet>
 */
import java.awt.*;
import java.awt.event.*;
import java.applet.*;

public class DoubleBuffer extends Applet {
    int gap = 3;
    int mx, my;
    boolean flicker = true;
    Image buffer = null;
    int w, h;

    public void init() {
        Dimension d = getSize();
        w = d.width;
        h = d.height;
        buffer = createImage(w, h);
        addMouseMotionListener(new MouseMotionAdapter() {
            public void mouseDragged(MouseEvent me) {
                mx = me.getX();
                my = me.getY();
                flicker = false;
                repaint();
            }
            public void mouseMoved(MouseEvent me) {
                mx = me.getX();
                my = me.getY();
                flicker = true;
            }
        });
    }
}
```

```
        repaint();
    }
});
}

public void paint(Graphics g) {
    Graphics screengc = null;

    if (!flicker) {
        screengc = g;
        g = buffer.getGraphics();
    }

    g.setColor(Color.blue);
    g.fillRect(0, 0, w, h);

    g.setColor(Color.red);
    for (int i=0; i<w; i+=gap)
        g.drawLine(i, 0, w-i, h);
    for (int i=0; i<h; i+=gap)
        g.drawLine(0, i, w, h-i);

    g.setColor(Color.black);
    g.drawString("Press mouse button to double buffer", 10, h/2);

    g.setColor(Color.yellow);
    g.fillOval(mx - gap, my - gap, gap*2+1, gap*2+1);

    if (!flicker) {
        screengc.drawImage(buffer, 0, 0, null);
    }
}

public void update(Graphics g) {
    paint(g);
}
}
```

This simple applet has a complicated `paint()` method. It fills the background with blue and then draws a red moiré pattern on top of that. It paints some black text on top of that and then paints a yellow circle centered at the coordinates `mx`, `my`. The `mouseMoved()` and `mouseDragged()` methods are overridden to track the mouse position. These methods are identical, except for the setting of the `flicker` Boolean variable. `mouseMoved()` sets `flicker` to `true`, and `mouseDragged()` sets it to `false`. This has the effect of calling `repaint()` with `flicker` set to `true` when the mouse is moved (but no button is pressed) and set to `false` when the mouse is dragged with any button pressed.

When `paint()` gets called with `flicker` set to `true`, we see each drawing operation as it is executed on the screen. In the case where a mouse button is pressed and `paint()` is called with `flicker` set to `false`, we see quite a different picture. The `paint()` method swaps the `Graphics` reference `g` with the graphics context that refers to the offscreen canvas, `buffer`, which we created in `init()`. Then all of the drawing operations are invisible. At the end of `paint()`, we simply call `drawImage()` to show the results of these drawing methods all at once.

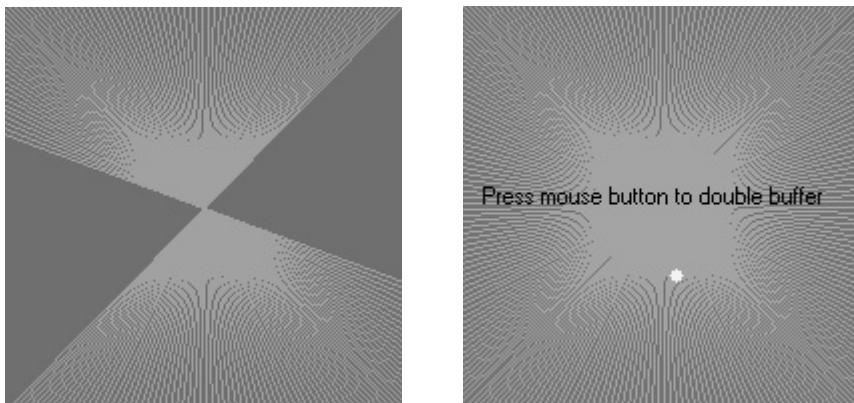


Figure 27-2 Output from **DoubleBuffer** without (left) and with (right) double buffering

Notice that it is okay to pass in a **null** as the fourth parameter to **drawImage()**. This is the parameter used to pass an **ImageObserver** object that receives notification of image events. Since this is an image that is not being produced from a network stream, we have no need for notification. The left snapshot in Figure 27-2 is what the applet looks like with the mouse button not pressed. As you can see, the image was in the middle of repainting when this snapshot was taken. The right snapshot shows how, when a mouse button is pressed, the image is always complete and clean due to double buffering.

MediaTracker

A **MediaTracker** is an object that will check the status of an arbitrary number of images in parallel. To use **MediaTracker**, you create a new instance and use its **addImage()** method to track the loading status of an image. **addImage()** has the following general forms:

```
void addImage(Image imgObj, int imgID)
void addImage(Image imgObj, int imgID, int width, int height)
```

Here, *imgObj* is the image being tracked. Its identification number is passed in *imgID*. ID numbers do not need to be unique. You can use the same number with several images as a means of identifying them as part of a group. Furthermore, images with lower IDs are given priority over those with higher IDs when loading. In the second form, *width* and *height* specify the dimensions of the object when it is displayed.

Once you've registered an image, you can check whether it's loaded, or you can wait for it to completely load. To check the status of an image, call **checkID()**. The version used in this chapter is shown here:

```
boolean checkID(int imgID)
```

Here, *imgID* specifies the ID of the image you want to check. The method returns **true** if all images that have the specified ID have been loaded (or if an error or user-abort has terminated loading). Otherwise, it returns **false**. You can use the **checkAll()** method to see if all images being tracked have been loaded.

You should use **MediaTracker** when loading a group of images. If all of the images that you're interested in aren't downloaded, you can display something else to entertain the user until they all arrive.

CAUTION If you use **MediaTracker** once you've called **addImage()** on an image, a reference in **MediaTracker** will prevent the system from garbage collecting it. If you want the system to be able to garbage collect images that were being tracked, make sure it can collect the **MediaTracker** instance as well.

Here's an example that loads a three-image slide show and displays a nice bar chart of the loading progress:

```
/*
 * <applet code="TrackedImageLoad" width=400 height=345>
 * <param name="img"
 * value="Lilies+SunFlower+ConeFlowers">
 * </applet>
 */
import java.util.*;
import java.applet.*;
import java.awt.*;

public class TrackedImageLoad extends Applet implements Runnable {
    MediaTracker tracker;
    int tracked;
    int frame_rate = 5;
    int current_img = 0;
    Thread motor;
    static final int MAXIMAGES = 10;
    Image img[] = new Image[MAXIMAGES];
    String name[] = new String[MAXIMAGES];
    volatile boolean stopFlag;

    public void init() {
        tracker = new MediaTracker(this);
        StringTokenizer st = new StringTokenizer(getParameter("img"),
                                              "+");
        while(st.hasMoreTokens() && tracked <= MAXIMAGES) {
            name[tracked] = st.nextToken();
            img[tracked] = getImage(getDocumentBase(),
                                   name[tracked] + ".jpg");
            tracker.addImage(img[tracked], tracked);
            tracked++;
        }
    }

    public void paint(Graphics g) {
        String loaded = "";
        int donecount = 0;

        for(int i=0; i<tracked; i++) {
            if (tracker.checkID(i, true)) {
```

```
        donecount++;
        loaded += name[i] + " ";
    }
}

Dimension d = getSize();
int w = d.width;
int h = d.height;

if (donecount == tracked) {
    frame_rate = 1;
    Image i = img[current_img++];
    int iw = i.getWidth(null);
    int ih = i.getHeight(null);
    g.drawImage(i, (w - iw)/2, (h - ih)/2, null);
    if (current_img >= tracked)
        current_img = 0;
} else {
    int x = w * donecount / tracked;
    g.setColor(Color.black);
    g.fillRect(0, h/3, x, 16);
    g.setColor(Color.white);
    g.fillRect(x, h/3, w-x, 16);
    g.setColor(Color.black);
    g.drawString(loaded, 10, h/2);
}
}

public void start() {
    motor = new Thread(this);
    stopFlag = false;
    motor.start();
}

public void stop() {
    stopFlag = true;
}

public void run() {
    motor.setPriority(Thread.MIN_PRIORITY);
    while (true) {
        repaint();
        try {
            Thread.sleep(1000/frame_rate);
        } catch (InterruptedException e) {
            System.out.println("Interrupted");
            return;
        }
        if (stopFlag)
            return;
    }
}
```

This example creates a new **MediaTracker** in the **init()** method and then adds each of the named images as a tracked image with **addImage()**. In the **paint()** method, it calls **checkID()** on each of the images that we're tracking. If all of the images are loaded, they are displayed. If not, a simple bar chart of the number of images loaded is shown, with the names of the fully loaded images displayed underneath the bar.

ImageProducer

ImageProducer is an interface for objects that want to produce data for images. An object that implements the **ImageProducer** interface will supply integer or byte arrays that represent image data and produce **Image** objects. As you saw earlier, one form of the **createImage()** method takes an **ImageProducer** object as its argument. There are two image producers contained in **java.awt.image**: **MemoryImageSource** and **FilteredImageSource**. Here, we will examine **MemoryImageSource** and create a new **Image** object from data generated in an applet.

MemoryImageSource

MemoryImageSource is a class that creates a new **Image** from an array of data. It defines several constructors. Here is the one we will be using:

```
MemoryImageSource(int width, int height, int pixel[], int offset,
                  int scanLineWidth)
```

The **MemoryImageSource** object is constructed out of the array of integers specified by *pixel*, in the default RGB color model to produce data for an **Image** object. In the default color model, a pixel is an integer with Alpha, Red, Green, and Blue (0xAARRGGBB). The Alpha value represents a degree of transparency for the pixel. Fully transparent is 0 and fully opaque is 255. The width and height of the resulting image are passed in *width* and *height*. The starting point in the pixel array to begin reading data is passed in *offset*. The width of a scan line (which is often the same as the width of the image) is passed in *scanLineWidth*.

The following short example generates a **MemoryImageSource** object using a variation on a simple algorithm (a bitwise-exclusive-OR of the x and y address of each pixel) from the book *Beyond Photography, The Digital Darkroom* by Gerard J. Holzmann (Prentice Hall, 1988).

```
/*
 * <applet code="MemoryImageGenerator" width=256 height=256>
 * </applet>
 */
import java.applet.*;
import java.awt.*;
import java.awt.image.*;

public class MemoryImageGenerator extends Applet {
    Image img;
    public void init() {
        Dimension d = getSize();
        int w = d.width;
```

```
int h = d.height;
int pixels[] = new int[w * h];
int i = 0;

for(int y=0; y<h; y++) {
    for(int x=0; x<w; x++) {
        int r = (x^y)&0xff;
        int g = (x*2^y*2)&0xff;
        int b = (x*4^y*4)&0xff;
        pixels[i++] = (255 << 24) | (r << 16) | (g << 8) | b;
    }
}
img = createImage(new MemoryImageSource(w, h, pixels, 0, w));
}

public void paint(Graphics g) {
    g.drawImage(img, 0, 0, this);
}
}
```

The data for the new **MemoryImageSource** is created in the **init()** method. An array of integers is created to hold the pixel values; the data is generated in the nested **for** loops where the **r**, **g**, and **b** values get shifted into a pixel in the **pixels** array. Finally, **createImage()** is called with a new instance of a **MemoryImageSource** created from the raw pixel data as its parameter. Figure 27-3 shows the image when we run the applet. (It looks much nicer in color.)

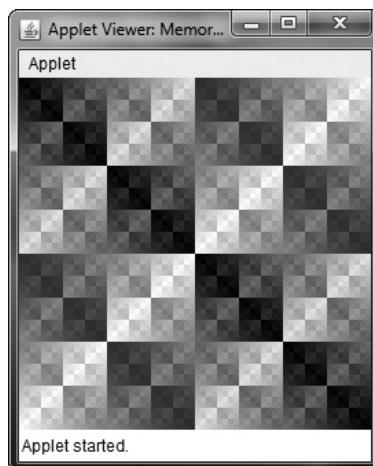


Figure 27-3 Sample output from **MemoryImageGenerator**

ImageConsumer

ImageConsumer is an interface for objects that want to take pixel data from images and supply it as another kind of data. This, obviously, is the opposite of **ImageProducer**, described earlier. An object that implements the **ImageConsumer** interface is going to create **int** or **byte** arrays that represent pixels from an **Image** object. We will examine the **PixelGrabber** class, which is a simple implementation of the **ImageConsumer** interface.

PixelGrabber

The **PixelGrabber** class is defined within **java.lang.image**. It is the inverse of the **MemoryImageSource** class. Rather than constructing an image from an array of pixel values, it takes an existing image and *grabs* the pixel array from it. To use **PixelGrabber**, you first create an array of **ints** big enough to hold the pixel data, and then you create a **PixelGrabber** instance passing in the rectangle that you want to grab. Finally, you call **grabPixels()** on that instance.

The **PixelGrabber** constructor that is used in this chapter is shown here:

```
PixelGrabber(Image imgObj, int left, int top, int width, int height, int pixel [ ],
            int offset, int scanLineWidth)
```

Here, *imgObj* is the object whose pixels are being grabbed. The values of *left* and *top* specify the upper-left corner of the rectangle, and *width* and *height* specify the dimensions of the rectangle from which the pixels will be obtained. The pixels will be stored in *pixel* beginning at *offset*. The width of a scan line (which is often the same as the width of the image) is passed in *scanLineWidth*.

grabPixels() is defined like this:

```
boolean grabPixels( )
    throws InterruptedException

boolean grabPixels(long milliseconds)
    throws InterruptedException
```

Both methods return **true** if successful and **false** otherwise. In the second form, *milliseconds* specifies how long the method will wait for the pixels. Both throw **InterruptedException** if execution is interrupted by another thread.

Here is an example that grabs the pixels from an image and then creates a histogram of pixel brightness. The *histogram* is simply a count of pixels that are a certain brightness for all brightness settings between 0 and 255. After the applet paints the image, it draws the histogram over the top.

```
/*
 * <applet code=HistoGrab width=400 height=345>
 * <param name=img value=Lilies.jpg>
 * </applet> */
import java.applet.*;
import java.awt.* ;
import java.awt.image.* ;
```

```
public class HistoGrab extends Applet {
    Dimension d;
    Image img;
    int iw, ih;
    int pixels[];
    int w, h;
    int hist[] = new int[256];
    int max_hist = 0;

    public void init() {
        d = getSize();
        w = d.width;
        h = d.height;

        try {
            img = getImage(getDocumentBase(), getParameter("img"));
            MediaTracker t = new MediaTracker(this);

            t.addImage(img, 0);
            t.waitForID(0);
            iw = img.getWidth(null);
            ih = img.getHeight(null);
            pixels = new int[iw * ih];
            PixelGrabber pg = new PixelGrabber(img, 0, 0, iw, ih,
                                              pixels, 0, iw);
            pg.grabPixels();
        } catch (InterruptedException e) {
            System.out.println("Interrupted");
            return;
        }

        for (int i=0; i<iw*ih; i++) {
            int p = pixels[i];
            int r = 0xff & (p >> 16);
            int g = 0xff & (p >> 8);
            int b = 0xff & (p);
            int y = (int) (.33 * r + .56 * g + .11 * b);
            hist[y]++;
        }
        for (int i=0; i<256; i++) {
            if (hist[i] > max_hist)
                max_hist = hist[i];
        }
    }

    public void update() {}

    public void paint(Graphics g) {
        g.drawImage(img, 0, 0, null);
        int x = (w - 256) / 2;
        int lasty = h - h * hist[0] / max_hist;
```

```
        for (int i=0; i<256; i++, x++) {
            int y = h - h * hist[i] / max_hist;
            g.setColor(new Color(i, i, i));
            g.fillRect(x, y, 1, h);
            g.setColor(Color.red);
            g.drawLine(x-1, lasty, x, y);
            lasty = y;
        }
    }
}
```

Figure 27-4 shows an example image and its histogram.

ImageFilter

Given the **ImageProducer** and **ImageConsumer** interface pair—and their concrete classes **MemoryImageSource** and **PixelGrabber**—you can create an arbitrary set of translation filters that takes a source of pixels, modifies them, and passes them on to an arbitrary consumer. This mechanism is analogous to the way concrete classes are created from the abstract I/O classes **InputStream**, **OutputStream**, **Reader**, and **Writer** (described in Chapter 20). This stream model for images is completed by the introduction of the **ImageFilter** class. Some subclasses of **ImageFilter** in the `java.awt.image` package are **AreaAveragingScaleFilter**, **CropImageFilter**, **ReplicateScaleFilter**, and **RGBImageFilter**. There is also an implementation of **ImageProducer** called **FilteredImageSource**, which takes an arbitrary **ImageFilter** and wraps it around an **ImageProducer** to filter the pixels it produces. An instance of **FilteredImageSource**



Figure 27-4 Sample output from **HistoGrab**

can be used as an **ImageProducer** in calls to **createImage()**, in much the same way that **BufferedInputStreams** can be passed off as **InputStreams**.

In this chapter, we examine two filters: **CropImageFilter** and **RGBImageFilter**.

CropImageFilter

CropImageFilter filters an image source to extract a rectangular region. One situation in which this filter is valuable is where you want to use several small images from a single, larger source image. Loading twenty 2K images takes much longer than loading a single 40K image that has many frames of an animation tiled into it. If every subimage is the same size, then you can easily extract these images by using **CropImageFilter** to disassemble the block once your program starts. Here is an example that creates 16 images taken from a single image. The tiles are then scrambled by swapping a random pair from the 16 images 32 times.

```
/*
 * <applet code=TileImage width=400 height=345>
 * <param name=img value=Lilies.jpg>
 * </applet>
 */
import java.applet.*;
import java.awt.*;
import java.awt.image.*;

public class TileImage extends Applet {
    Image img;
    Image cell[] = new Image[4*4];
    int iw, ih;
    int tw, th;

    public void init() {
        try {
            img = getImage(getDocumentBase(), getParameter("img"));
            MediaTracker t = new MediaTracker(this);
            t.addImage(img, 0);
            t.waitForID(0);
            iw = img.getWidth(null);
            ih = img.getHeight(null);
            tw = iw / 4;
            th = ih / 4;
            CropImageFilter f;
            FilteredImageSource fis;
            t = new MediaTracker(this);
            for (int y=0; y<4; y++) {
                for (int x=0; x<4; x++) {
                    f = new CropImageFilter(tw*x, th*y, tw, th);
                    fis = new FilteredImageSource(img.getSource(), f);
                    int i = y*4+x;
                    cell[i] = createImage(fis);
                    t.addImage(cell[i], i);
                }
            }
        }
    }
}
```

```
        }
        t.waitForAll();
        for (int i=0; i<32; i++) {
            int si = (int)(Math.random() * 16);
            int di = (int)(Math.random() * 16);
            Image tmp = cell[si];
            cell[si] = cell[di];
            cell[di] = tmp;
        }
    } catch (InterruptedException e) {
    System.out.println("Interrupted");
}
}

public void update(Graphics g) {
    paint(g);
}

public void paint(Graphics g) {
    for (int y=0; y<4; y++) {
        for (int x=0; x<4; x++) {
            g.drawImage(cell[y*4+x], x * tw, y * th, null);
        }
    }
}
```

Figure 27-5 shows the flowers image scrambled by the **TileImage** applet.



Figure 27-5 Sample output from **TileImage**

RGBImageFilter

The **RGBImageFilter** is used to convert one image to another, pixel by pixel, transforming the colors along the way. This filter could be used to brighten an image, to increase its contrast, or even to convert it to grayscale.

To demonstrate **RGBImageFilter**, we have developed a somewhat complicated example that employs a dynamic plug-in strategy for image-processing filters. We've created an interface for generalized image filtering so that an applet can simply load these filters based on **<param>** tags without having to know about all of the **ImageFilters** in advance. This example consists of the main applet class called **ImageFilterDemo**, the interface called **PlugInFilter**, and a utility class called **LoadedImage**, which encapsulates some of the **MediaTracker** methods we've been using in this chapter. Also included are three filters—**Grayscale**, **Invert**, and **Contrast**—which simply manipulate the color space of the source image using **RGBImageFilters**, and two more classes—**Blur** and **Sharpen**—which do more complicated "convolution" filters that change pixel data based on the pixels surrounding each pixel of source data. **Blur** and **Sharpen** are subclasses of an abstract helper class called **Convolver**. Let's look at each part of our example.

ImageFilterDemo.java

The **ImageFilterDemo** class is the applet framework for our sample image filters. It employs a simple **BorderLayout**, with a **Panel** at the *South* position to hold the buttons that will represent each filter. A **Label** object occupies the *North* slot for informational messages about filter progress. The *Center* is where the image (which is encapsulated in the **LoadedImage** **Canvas** subclass, described later) is put. We parse the buttons/filters out of the filters **<param>** tag, separating them with +'s using a **StringTokenizer**.

The **actionPerformed()** method is interesting because it uses the label from a button as the name of a filter class that it tries to load with (**PlugInFilter**) **Class.forName(a).newInstance()**. This method is robust and takes appropriate action if the button does not correspond to a proper class that implements **PlugInFilter**.

```
/*
 * <applet code=ImageFilterDemo width=400 height=345>
 * <param name=img value=Lilies.jpg>
 * <param name=filters value="Grayscale+Invert+Contrast+Blur+Sharpen">
 * </applet>
 */
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;

public class ImageFilterDemo extends Applet implements ActionListener {
    Image img;
    PlugInFilter pif;
    Image fimg;
    Image curImg;
    LoadedImage lim;
    Label lab;
    Button reset;
```

```
public void init() {
    setLayout(new BorderLayout());
    Panel p = new Panel();

    add(p, BorderLayout.SOUTH);
    reset = new Button("Reset");
    reset.addActionListener(this);
    p.add(reset);
    StringTokenizer st = new StringTokenizer(getParameter("filters"), "+");

    while(st.hasMoreTokens()) {
        Button b = new Button(st.nextToken());
        b.addActionListener(this);
        p.add(b);
    }

    lab = new Label("");
    add(lab, BorderLayout.NORTH);

    img = getImage(getDocumentBase(), getParameter("img"));
    lim = new LoadedImage(img);
    add(lim, BorderLayout.CENTER);
}

public void actionPerformed(ActionEvent ae) {
    String a = "";
    try {
        a = ae.getActionCommand();
        if (a.equals("Reset")) {
            lim.set(img);
            lab.setText("Normal");
        }
        else {
            pif = (PlugInFilter) Class.forName(a).newInstance();
            fimg = pif.filter(this, img);
            lim.set(fimg);
            lab.setText("Filtered: " + a);
        }
        repaint();
    } catch (ClassNotFoundException e) {
        lab.setText(a + " not found");
        lim.set(img);
        repaint();
    } catch (InstantiationException e) {
        lab.setText("couldn't new " + a);
    } catch (IllegalAccessException e) {
        lab.setText("no access: " + a);
    }
}
```

Figure 27-6 shows what the applet looks like when it is first loaded using the applet tag shown at the top of this source file.



Figure 27-6 Sample normal output from **ImageFilterDemo**

PlugInFilter.java

PlugInFilter is a simple interface used to abstract image filtering. It has only one method, **filter()**, which takes the applet and the source image and returns a new image that has been filtered in some way.

```
interface PlugInFilter {
    java.awt.Image filter(java.applet.Applet a, java.awt.Image in);
}
```

LoadedImage.java

LoadedImage is a convenient subclass of **Canvas**, which takes an image at construction time and synchronously loads it using **MediaTracker**. **LoadedImage** then behaves properly inside of **LayoutManager** control, because it overrides the **getPreferredSize()** and **getMinimumSize()** methods. Also, it has a method called **set()** that can be used to set a new **Image** to be displayed in this **Canvas**. That is how the filtered image is displayed after the plug-in is finished.

```
import java.awt.*;
public class LoadedImage extends Canvas {
    Image img;
    public LoadedImage(Image i) {
        set(i);
    }
    void set(Image i) {
        MediaTracker mt = new MediaTracker(this);
    }
}
```

```

        mt.addImage(i, 0);
        try {
            mt.waitForAll();
        } catch (InterruptedException e) {
            System.out.println("Interrupted");
            return;
        }
        img = i;
        repaint();
    }

    public void paint(Graphics g) {
        if (img == null) {
            g.drawString("no image", 10, 30);
        } else {
            g.drawImage(img, 0, 0, this);
        }
    }

    public Dimension getPreferredSize() {
        return new Dimension(img.getWidth(this), img.getHeight(this));
    }

    public Dimension getMinimumSize() {
        return getPreferredSize();
    }
}

```

Grayscale.java

The **Grayscale** filter is a subclass of **RGBImageFilter**, which means that **Grayscale** can use itself as the **ImageFilter** parameter to **FilteredImageSource**'s constructor. Then all it needs to do is override **filterRGB()** to change the incoming color values. It takes the red, green, and blue values and computes the brightness of the pixel, using the NTSC (National Television Standards Committee) color-to-brightness conversion factor. It then simply returns a gray pixel that is the same brightness as the color source.

```

import java.applet.*;
import java.awt.*;
import java.awt.image.*;

class Grayscale extends RGBImageFilter implements PlugInFilter {
    public Image filter(Applet a, Image in) {
        return a.createImage(new FilteredImageSource(in.getSource(), this));
    }

    public int filterRGB(int x, int y, int rgb) {
        int r = (rgb >> 16) & 0xff;
        int g = (rgb >> 8) & 0xff;
        int b = rgb & 0xff;
        int k = (int) (.56 * g + .33 * r + .11 * b);
        return (0xff000000 | k << 16 | k << 8 | k);
    }
}

```

Invert.java

The **Invert** filter is also quite simple. It takes apart the red, green, and blue channels and then inverts them by subtracting them from 255. These inverted values are packed back into a pixel value and returned.

```
import java.applet.*;
import java.awt.*;
import java.awt.image.*;

class Invert extends RGBImageFilter implements PlugInFilter {
    public Image filter(Applet a, Image in) {
        return a.createImage(new FilteredImageSource(in.getSource(), this));
    }

    public int filterRGB(int x, int y, int rgb) {
        int r = 0xff - (rgb >> 16) & 0xff;
        int g = 0xff - (rgb >> 8) & 0xff;
        int b = 0xff - rgb & 0xff;
        return (0xff000000 | r << 16 | g << 8 | b);
    }
}
```

Figure 27-7 shows the image after it has been run through the **Invert** filter.

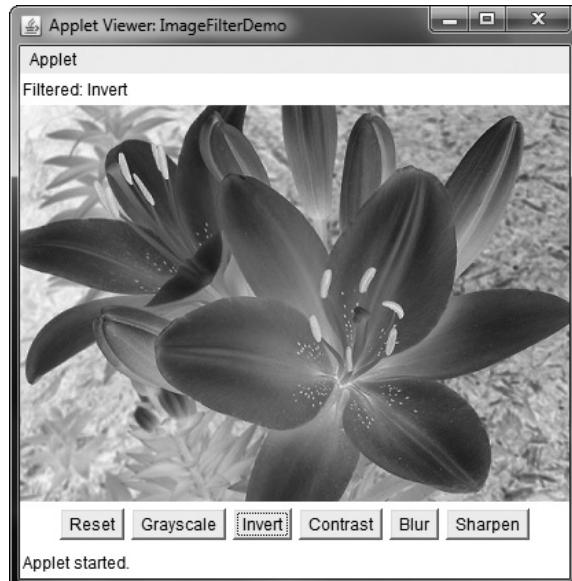


Figure 27-7 Using the **Invert** filter with **ImageFilterDemo**

Contrast.java

The **Contrast** filter is very similar to **Grayscale**, except its override of **filterRGB()** is slightly more complicated. The algorithm it uses for contrast enhancement takes the red, green, and blue values separately and boosts them by 1.2 times if they are already brighter than 128. If they are below 128, then they are divided by 1.2. The boosted values are properly clamped at 255 by the **multclamp()** method.

```
import java.applet.*;
import java.awt.*;
import java.awt.image.*;

public class Contrast extends RGBImageFilter implements PlugInFilter {

    public Image filter(Applet a, Image in) {
        return a.createImage(new FilteredImageSource(in.getSource(), this));
    }

    private int multclamp(int in, double factor) {
        in = (int) (in * factor);
        return in > 255 ? 255 : in;
    }

    double gain = 1.2;
    private int cont(int in) {
        return (in < 128) ? (int)(in/gain) : multclamp(in, gain);
    }

    public int filterRGB(int x, int y, int rgb) {
        int r = cont((rgb >> 16) & 0xff);
        int g = cont((rgb >> 8) & 0xff);
        int b = cont(rgb & 0xff);
        return (0xff000000 | r << 16 | g << 8 | b);
    }
}
```

Figure 27-8 shows the image after **Contrast** is pressed.

Convolver.java

The abstract class **Convolver** handles the basics of a convolution filter by implementing the **ImageConsumer** interface to move the source pixels into an array called **imgpixels**. It also creates a second array called **newimgpixels** for the filtered data. Convolution filters sample a small rectangle of pixels around each pixel in an image, called the *convolution kernel*. This area, 3×3 pixels in this demo, is used to decide how to change the center pixel in the area.

NOTE The reason that the filter can't modify the **imgpixels** array in place is that the next pixel on a scan line would try to use the original value for the previous pixel, which would have just been filtered away.

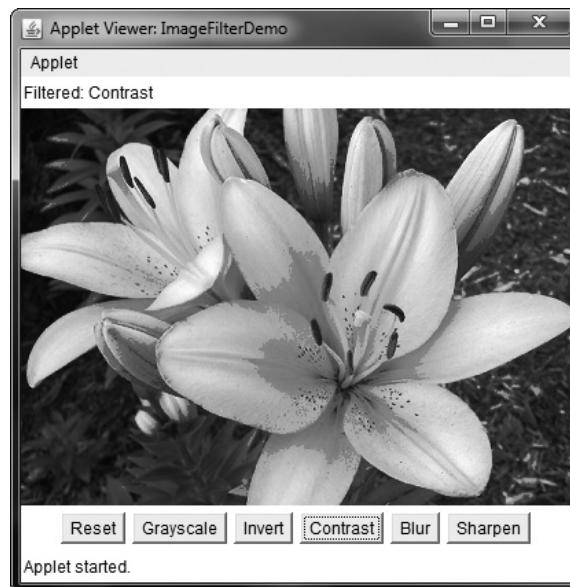


Figure 27-8 Using the **Contrast** filter with **ImageFilterDemo**

The two concrete subclasses, shown in the next section, simply implement the **convolve()** method, using **imgpixels** for source data and **newimgpixels** to store the result.

```
import java.applet.*;
import java.awt.*;
import java.awt.image.*;

abstract class Convolver implements ImageConsumer, PlugInFilter {
    int width, height;
    int imgpixels[], newimgpixels[];
    boolean imageReady = false;

    abstract void convolve(); // filter goes here...

    public Image filter(Applet a, Image in) {
        imageReady = false;
        in.getSource().startProduction(this);

        waitForImage();
        newimgpixels = new int[width*height];

        try {
            convolve();
        } catch (Exception e) {
            System.out.println("Convolver failed: " + e);
            e.printStackTrace();
        }
    }
}
```

```
        return a.createImage(
            new MemoryImageSource(width, height, newimgpixels, 0, width));
    }

    synchronized void waitForImage() {
        try {
            while(!imageReady) wait();
        } catch (Exception e) {
            System.out.println("Interrupted");
        }
    }

    public void setProperties(java.util.Hashtable<?,?> dummy) { }
    public void setColorModel(ColorModel dummy) { }
    public void setHints(int dummy) { }

    public synchronized void imageComplete(int dummy) {
        imageReady = true;
        notifyAll();
    }

    public void setDimensions(int x, int y) {
        width = x;
        height = y;
        imgpixels = new int[x*y];
    }

    public void setPixels(int x1, int y1, int w, int h,
        ColorModel model, byte pixels[], int off, int scansize) {
        int pix, x, y, x2, y2, sx, sy;

        x2 = x1+w;
        y2 = y1+h;
        sy = off;
        for(y=y1; y<y2; y++) {
            sx = sy;
            for(x=x1; x<x2; x++) {
                pix = model.getRGB(pixels[sx++]);
                if((pix & 0xff000000) == 0)
                    pix = 0xffffffff;
                imgpixels[y*width+x] = pix;
            }
            sy += scansize;
        }
    }

    public void setPixels(int x1, int y1, int w, int h,
        ColorModel model, int pixels[], int off, int scansize) {
        int pix, x, y, x2, y2, sx, sy;

        x2 = x1+w;
        y2 = y1+h;
        sy = off;
```

```

        for(y=y1; y<y2; y++) {
            sx = sy;
            for(x=x1; x<x2; x++) {
                pix = model.getRGB(pixels[sx++]);
                if((pix & 0xff000000) == 0)
                    pix = 0x00ffff;
                imgpixels[y*width+x] = pix;
            }
            sy += scansize;
        }
    }
}

```

NOTE A built-in convolution filter called **ConvolveOp** is provided by **java.awt.image**. You may want to explore its capabilities on your own.

Blur.java

The **Blur** filter is a subclass of **Convolver** and simply runs through every pixel in the source image array, **imgpixels**, and computes the average of the 3×3 box surrounding it. The corresponding output pixel in **newimgpixels** is that average value.

```

public class Blur extends Convolver {
    public void convolve() {
        for(int y=1; y<height-1; y++) {
            for(int x=1; x<width-1; x++) {
                int rs = 0;
                int gs = 0;
                int bs = 0;

                for(int k=-1; k<=1; k++) {
                    for(int j=-1; j<=1; j++) {
                        int rgb = imgpixels[(y+k)*width+x+j];
                        int r = (rgb >> 16) & 0xff;
                        int g = (rgb >> 8) & 0xff;
                        int b = rgb & 0xff;
                        rs += r;
                        gs += g;
                        bs += b;
                    }
                }
                rs /= 9;
                gs /= 9;
                bs /= 9;

                newimgpixels[y*width+x] = (0xff000000 |
                                              rs << 16 | gs << 8 | bs);
            }
        }
    }
}

```

Figure 27-9 shows the applet after **Blur**.



Figure 27-9 Using the **Blur** filter with **ImageFilterDemo**

Sharpen.java

The **Sharpen** filter is also a subclass of **Convolver** and is (more or less) the inverse of **Blur**. It runs through every pixel in the source image array, **imgpixels**, and computes the average of the 3×3 box surrounding it, not counting the center. The corresponding output pixel in **newimgpixels** has the difference between the center pixel and the surrounding average added to it. This basically says that if a pixel is 30 brighter than its surroundings, make it another 30 brighter. If, however, it is 10 darker, then make it another 10 darker. This tends to accentuate edges while leaving smooth areas unchanged.

```
public class Sharpen extends Convolver {  
  
    private final int clamp(int c) {  
        return (c > 255 ? 255 : (c < 0 ? 0 : c));  
    }  
  
    public void convolve() {  
        int r0=0, g0=0, b0=0;  
  
        for(int y=1; y<height-1; y++) {  
            for(int x=1; x<width-1; x++) {  
                int rs = 0;  
                int gs = 0;  
                int bs = 0;  
  
                for(int k=-1; k<=1; k++) {  
                    for(int j=-1; j<=1; j++) {  
                        rs += imgpixels[(y+k)*width+x+j];  
                        gs += imgpixels[(y+k)*width+x+j];  
                        bs += imgpixels[(y+k)*width+x+j];  
                    }  
                }  
                rs /= 9;  
                gs /= 9;  
                bs /= 9;  
  
                newimgpixels[y*width+x] = clamp(c + (rs - c) * 3);  
                newimgpixels[y*width+x+1] = clamp(c + (gs - c) * 3);  
                newimgpixels[y*width+x+2] = clamp(c + (bs - c) * 3);  
            }  
        }  
    }  
}
```

```
int rgb = imgpixels[(y+k)*width+x+j];
int r = (rgb >> 16) & 0xff;
int g = (rgb >> 8) & 0xff;
int b = rgb & 0xff;
if (j == 0 && k == 0) {
    r0 = r;
    g0 = g;
    b0 = b;
} else {
    rs += r;
    gs += g;
    bs += b;
}
}

rs >>= 3;
gs >>= 3;
bs >>= 3;
newimgpixels[y*width+x] = (0xff000000 |
    clamp(r0+r0-rs) << 16 |
    clamp(g0+g0-gs) << 8 |
    clamp(b0+b0-bs));
}
}
}
}
```

Figure 27-10 shows the applet after **Sharpen**.



Figure 27-10 Using the **Sharpen** filter with **ImageFilterDemo**

Additional Imaging Classes

In addition to the imaging classes described in this chapter, **java.awt.image** supplies several others that offer enhanced control over the imaging process and that support advanced imaging techniques. Also available is the imaging package called **javax.imageio**. This package supports plug-ins that handle various image formats. If sophisticated graphical output is of special interest to you, then you will want to explore the additional classes found in **java.awt.image** and **javax.imageio**.

This page has been intentionally left blank

CHAPTER

28

The Concurrency Utilities

From the start, Java has provided built-in support for multithreading and synchronization. For example, new threads can be created by implementing **Runnable** or by extending **Thread**; synchronization is available by use of the **synchronized** keyword; and interthread communication is supported by the **wait()** and **notify()** methods that are defined by **Object**. In general, this built-in support for multithreading was one of Java’s most important innovations and is still one of its major strengths.

However, as conceptually pure as Java’s original support for multithreading is, it is not ideal for all applications—especially those that make intensive use of multiple threads. For example, the original multithreading support does not provide several high-level features, such as semaphores, thread pools, and execution managers, that facilitate the creation of intensively concurrent programs.

It is important to explain at the outset that many Java programs make use of multithreading and are, therefore, “concurrent.” For example, many applets and servlets use multithreading. However, as it is used in this chapter, the term *concurrent program* refers to a program that makes *extensive, integral* use of concurrently executing threads. An example of such a program is one that uses separate threads to simultaneously compute the partial results of a larger computation. Another example is a program that coordinates the activities of several threads, each of which seeks access to information in a database. In this case, read-only accesses might be handled differently from those that require read/write capabilities.

To begin to handle the needs of a concurrent program, JDK 5 added the *concurrency utilities*, also commonly referred to as the *concurrent API*. The original set of concurrency utilities supplied many features that had long been wanted by programmers who develop concurrent applications. For example, it offered synchronizers (such as the semaphore), thread pools, execution managers, locks, several concurrent collections, and a streamlined way to use threads to obtain computational results.

Although the original concurrent API was impressive in its own right, it was significantly expanded by JDK 7. The most important addition was the *Fork/Join Framework*. The Fork/Join Framework facilitates the creation of programs that make use of multiple processors (such as those found in multicore systems). Thus, it streamlines the development of programs in

which two or more pieces execute with true simultaneity (that is, true parallel execution), not just time-slicing. As you can easily imagine, parallel execution can dramatically increase the speed of certain operations. Because multicore systems are now commonplace, the inclusion of the Fork/Join Framework was as timely as it was powerful. With the release of JDK 8, the Fork/Join Framework was further enhanced.

In addition, JDK 8 included some new features related to other parts of the concurrent API. Thus, the concurrent API continues to evolve and expand to meet the needs of the contemporary computing environment.

The original concurrent API was quite large, and the additions made by JDK 7 and JDK 8 have increased its size substantially. As you might expect, many of the issues surrounding the concurrency utilities are quite complex. It is beyond the scope of this book to discuss all of its facets. The preceding notwithstanding, it is important for all programmers to have a general, working knowledge of key aspects of the concurrent API. Even in programs that are not intensively parallel, features such as synchronizers, callable threads, and executors, are applicable to a wide variety of situations. Perhaps most importantly, because of the rise of multicore computers, solutions involving the Fork/Join Framework are becoming more common. For these reasons, this chapter presents an overview of several core features defined by the concurrency utilities and shows a number of examples that demonstrate their use. It concludes with an introduction to the Fork/Join Framework.

The Concurrent API Packages

The concurrency utilities are contained in the **java.util.concurrent** package and in its two subpackages: **java.util.concurrent.atomic** and **java.util.concurrent.locks**. A brief overview of their contents is given here.

java.util.concurrent

java.util.concurrent defines the core features that support alternatives to the built-in approaches to synchronization and interthread communication. It defines the following key features:

- Synchronizers
- Executors
- Concurrent collections
- The Fork/Join Framework

Synchronizers offer high-level ways of synchronizing the interactions between multiple threads. The synchronizer classes defined by **java.util.concurrent** are

Semaphore	Implements the classic semaphore.
CountDownLatch	Waits until a specified number of events have occurred.
CyclicBarrier	Enables a group of threads to wait at a predefined execution point.
Exchanger	Exchanges data between two threads.
Phaser	Synchronizes threads that advance through multiple phases of an operation.

Notice that each synchronizer provides a solution to a specific type of synchronization problem. This enables each synchronizer to be optimized for its intended use. In the past, these types of synchronization objects had to be crafted by hand. The concurrent API standardizes them and makes them available to all Java programmers.

Executors manage thread execution. At the top of the executor hierarchy is the **Executor** interface, which is used to initiate a thread. **ExecutorService** extends **Executor** and provides methods that manage execution. There are three implementations of **ExecutorService**: **ThreadPoolExecutor**, **ScheduledThreadPoolExecutor**, and **ForkJoinPool**. `java.util.concurrent` also defines the **Executors** utility class, which includes a number of static methods that simplify the creation of various executors.

Related to executors are the **Future** and **Callable** interfaces. A **Future** contains a value that is returned by a thread after it executes. Thus, its value becomes defined “in the future,” when the thread terminates. **Callable** defines a thread that returns a value.

`java.util.concurrent` defines several concurrent collection classes, including **ConcurrentHashMap**, **ConcurrentLinkedQueue**, and **CopyOnWriteArrayList**. These offer concurrent alternatives to their related classes defined by the Collections Framework.

The *Fork/Join Framework* supports parallel programming. Its main classes are **ForkJoinTask**, **ForkJoinPool**, **RecursiveTask**, and **RecursiveAction**.

Finally, to better handle thread timing, `java.util.concurrent` defines the **TimeUnit** enumeration.

java.util.concurrent.atomic

`java.util.concurrent.atomic` facilitates the use of variables in a concurrent environment. It provides a means of efficiently updating the value of a variable without the use of locks. This is accomplished through the use of classes, such as **AtomicInteger** and **AtomicLong**, and methods, such as **compareAndSet()**, **decrementAndGet()**, and **getAndSet()**. These methods execute as a single, non-interruptible operation.

java.util.concurrent.locks

`java.util.concurrent.locks` provides an alternative to the use of synchronized methods. At the core of this alternative is the **Lock** interface, which defines the basic mechanism used to acquire and relinquish access to an object. The key methods are **lock()**, **tryLock()**, and **unlock()**. The advantage to using these methods is greater control over synchronization.

The remainder of this chapter takes a closer look at the constituents of the concurrent API.

Using Synchronization Objects

Synchronization objects are supported by the **Semaphore**, **CountDownLatch**, **CyclicBarrier**, **Exchanger**, and **Phaser** classes. Collectively, they enable you to handle several formerly difficult synchronization situations with ease. They are also applicable to a wide range of programs—even those that contain only limited concurrency. Because the synchronization objects will be of interest to nearly all Java programs, each is examined here in some detail.

Semaphore

The synchronization object that many readers will immediately recognize is **Semaphore**, which implements a classic semaphore. A semaphore controls access to a shared resource through the use of a counter. If the counter is greater than zero, then access is allowed. If it is zero, then access is denied. What the counter is counting are *permits* that allow access to the shared resource. Thus, to access the resource, a thread must be granted a permit from the semaphore.

In general, to use a semaphore, the thread that wants access to the shared resource tries to acquire a permit. If the semaphore's count is greater than zero, then the thread acquires a permit, which causes the semaphore's count to be decremented. Otherwise, the thread will be blocked until a permit can be acquired. When the thread no longer needs access to the shared resource, it releases the permit, which causes the semaphore's count to be incremented. If there is another thread waiting for a permit, then that thread will acquire a permit at that time. Java's **Semaphore** class implements this mechanism.

Semaphore has the two constructors shown here:

```
Semaphore(int num)
Semaphore(int num, boolean how)
```

Here, *num* specifies the initial permit count. Thus, *num* specifies the number of threads that can access a shared resource at any one time. If *num* is one, then only one thread can access the resource at any one time. By default, waiting threads are granted a permit in an undefined order. By setting *how* to **true**, you can ensure that waiting threads are granted a permit in the order in which they requested access.

To acquire a permit, call the **acquire()** method, which has these two forms:

```
void acquire() throws InterruptedException
void acquire(int num) throws InterruptedException
```

The first form acquires one permit. The second form acquires *num* permits. Most often, the first form is used. If the permit cannot be granted at the time of the call, then the invoking thread suspends until the permit is available.

To release a permit, call **release()**, which has these two forms:

```
void release()
void release(int num)
```

The first form releases one permit. The second form releases the number of permits specified by *num*.

To use a semaphore to control access to a resource, each thread that wants to use that resource must first call **acquire()** before accessing the resource. When the thread is done with the resource, it must call **release()**. Here is an example that illustrates the use of a semaphore:

```
// A simple semaphore example.

import java.util.concurrent.*;

class SemDemo {
```

```
public static void main(String args[]) {
    Semaphore sem = new Semaphore(1);

    new IncThread(sem, "A");
    new DecThread(sem, "B");

}

// A shared resource.
class Shared {
    static int count = 0;
}

// A thread of execution that increments count.
class IncThread implements Runnable {
    String name;
    Semaphore sem;

    IncThread(Semaphore s, String n) {
        sem = s;
        name = n;
        new Thread(this).start();
    }

    public void run() {

        System.out.println("Starting " + name);

        try {
            // First, get a permit.
            System.out.println(name + " is waiting for a permit.");
            sem.acquire();
            System.out.println(name + " gets a permit.");

            // Now, access shared resource.
            for(int i=0; i < 5; i++) {
                Shared.count++;
                System.out.println(name + ": " + Shared.count);

                // Now, allow a context switch -- if possible.
                Thread.sleep(10);
            }
        } catch (InterruptedException exc) {
            System.out.println(exc);
        }

        // Release the permit.
        System.out.println(name + " releases the permit.");
        sem.release();
    }
}
```

```

// A thread of execution that decrements count.
class DecThread implements Runnable {
    String name;
    Semaphore sem;

    DecThread(Semaphore s, String n) {
        sem = s;
        name = n;
        new Thread(this).start();
    }

    public void run() {

        System.out.println("Starting " + name);

        try {
            // First, get a permit.
            System.out.println(name + " is waiting for a permit.");
            sem.acquire();
            System.out.println(name + " gets a permit.");

            // Now, access shared resource.
            for(int i=0; i < 5; i++) {
                Shared.count--;
                System.out.println(name + ": " + Shared.count);

                // Now, allow a context switch -- if possible.
                Thread.sleep(10);
            }
        } catch (InterruptedException exc) {
            System.out.println(exc);
        }

        // Release the permit.
        System.out.println(name + " releases the permit.");
        sem.release();
    }
}

```

The output from the program is shown here. (The precise order in which the threads execute may vary.)

```

Starting A
A is waiting for a permit.
A gets a permit.
A: 1
Starting B
B is waiting for a permit.
A: 2
A: 3
A: 4
A: 5
A releases the permit.
B gets a permit.

```

```
B: 4
B: 3
B: 2
B: 1
B: 0
B releases the permit.
```

The program uses a semaphore to control access to the **count** variable, which is a static variable within the **Shared** class. **Shared.count** is incremented five times by the **run()** method of **IncThread** and decremented five times by **DecThread**. To prevent these two threads from accessing **Shared.count** at the same time, access is allowed only after a permit is acquired from the controlling semaphore. After access is complete, the permit is released. In this way, only one thread at a time will access **Shared.count**, as the output shows.

In both **IncThread** and **DecThread**, notice the call to **sleep()** within **run()**. It is used to “prove” that accesses to **Shared.count** are synchronized by the semaphore. In **run()**, the call to **sleep()** causes the invoking thread to pause between each access to **Shared.count**. This would normally enable the second thread to run. However, because of the semaphore, the second thread must wait until the first has released the permit, which happens only after all accesses by the first thread are complete. Thus, **Shared.count** is first incremented five times by **IncThread** and then decremented five times by **DecThread**. The increments and decrements are *not* intermixed.

Without the use of the semaphore, accesses to **Shared.count** by both threads would have occurred simultaneously, and the increments and decrements would be intermixed. To confirm this, try commenting out the calls to **acquire()** and **release()**. When you run the program, you will see that access to **Shared.count** is no longer synchronized, and each thread accesses it as soon as it gets a timeslice.

Although many uses of a semaphore are as straightforward as that shown in the preceding program, more intriguing uses are also possible. Here is an example. The following program reworks the producer/consumer program shown in Chapter 11 so that it uses two semaphores to regulate the producer and consumer threads, ensuring that each call to **put()** is followed by a corresponding call to **get()**:

```
// An implementation of a producer and consumer
// that use semaphores to control synchronization.

import java.util.concurrent.Semaphore;

class Q {
    int n;

    // Start with consumer semaphore unavailable.
    static Semaphore semCon = new Semaphore(0);
    static Semaphore semProd = new Semaphore(1);

    void get() {
        try {
            semCon.acquire();
        } catch(InterruptedException e) {
            System.out.println("InterruptedException caught");
        }
    }

    void put() {
        try {
            semProd.release();
        } catch(InterruptedException e) {
            System.out.println("InterruptedException caught");
        }
    }
}
```

```
        System.out.println("Got: " + n);
        semProd.release();
    }

    void put(int n) {
        try {
            semProd.acquire();
        } catch(InterruptedException e) {
            System.out.println("InterruptedException caught");
        }

        this.n = n;
        System.out.println("Put: " + n);
        semCon.release();
    }
}

class Producer implements Runnable {
    Q q;

    Producer(Q q) {
        this.q = q;
        new Thread(this, "Producer").start();
    }

    public void run() {
        for(int i=0; i < 20; i++) q.put(i);
    }
}

class Consumer implements Runnable {
    Q q;

    Consumer(Q q) {
        this.q = q;
        new Thread(this, "Consumer").start();
    }

    public void run() {
        for(int i=0; i < 20; i++) q.get();
    }
}

class ProdCon {
    public static void main(String args[]) {
        Q q = new Q();
        new Consumer(q);
        new Producer(q);
    }
}
```

A portion of the output is shown here:

```
Put: 0
Got: 0
Put: 1
Got: 1
Put: 2
Got: 2
Put: 3
Got: 3
Put: 4
Got: 4
Put: 5
Got: 5
.
.
.
```

As you can see, the calls to `put()` and `get()` are synchronized. That is, each call to `put()` is followed by a call to `get()` and no values are missed. Without the semaphores, multiple calls to `put()` would have occurred without matching calls to `get()`, resulting in values being missed. (To prove this, remove the semaphore code and observe the results.)

The sequencing of `put()` and `get()` calls is handled by two semaphores: `semProd` and `semCon`. Before `put()` can produce a value, it must acquire a permit from `semProd`. After it has set the value, it releases `semCon`. Before `get()` can consume a value, it must acquire a permit from `semCon`. After it consumes the value, it releases `semProd`. This “give and take” mechanism ensures that each call to `put()` must be followed by a call to `get()`.

Notice that `semCon` is initialized with no available permits. This ensures that `put()` executes first. The ability to set the initial synchronization state is one of the more powerful aspects of a semaphore.

CountDownLatch

Sometimes you will want a thread to wait until one or more events have occurred. To handle such a situation, the concurrent API supplies `CountDownLatch`. A `CountDownLatch` is initially created with a count of the number of events that must occur before the latch is released. Each time an event happens, the count is decremented. When the count reaches zero, the latch opens.

`CountDownLatch` has the following constructor:

```
CountDownLatch(int num)
```

Here, `num` specifies the number of events that must occur in order for the latch to open.

To wait on the latch, a thread calls `await()`, which has the forms shown here:

```
void await() throws InterruptedException
boolean await(long wait, TimeUnit tu) throws InterruptedException
```

The first form waits until the count associated with the invoking `CountDownLatch` reaches zero. The second form waits only for the period of time specified by `wait`. The units represented by `wait` are specified by `tu`, which is an object the `TimeUnit` enumeration.

(**TimeUnit** is described later in this chapter.) It returns **false** if the time limit is reached and **true** if the countdown reaches zero.

To signal an event, call the **countDown()** method, shown next:

```
void countDown()
```

Each call to **countDown()** decrements the count associated with the invoking object.

The following program demonstrates **CountDownLatch**. It creates a latch that requires five events to occur before it opens.

```
// An example of CountDownLatch.

import java.util.concurrent.CountDownLatch;

class CDLDemo {
    public static void main(String args[]) {
        CountDownLatch cdl = new CountDownLatch(5);

        System.out.println("Starting");

        new MyThread(cdl);

        try {
            cdl.await();
        } catch (InterruptedException exc) {
            System.out.println(exc);
        }

        System.out.println("Done");
    }
}

class MyThread implements Runnable {
    CountDownLatch latch;

    MyThread(CountDownLatch c) {
        latch = c;
        new Thread(this).start();
    }

    public void run() {
        for(int i = 0; i<5; i++) {
            System.out.println(i);
            latch.countDown(); // decrement count
        }
    }
}
```

The output produced by the program is shown here:

```
Starting
0
1
```

```

2
3
4
Done

```

Inside `main()`, a **CountDownLatch** called `cld` is created with an initial count of five. Next, an instance of **MyThread** is created, which begins execution of a new thread. Notice that `cld` is passed as a parameter to **MyThread**'s constructor and stored in the **latch** instance variable. Then, the main thread calls `await()` on `cld`, which causes execution of the main thread to pause until `cld`'s count has been decremented five times.

Inside the `run()` method of **MyThread**, a loop is created that iterates five times. With each iteration, the `countDown()` method is called on `latch`, which refers to `cld` in `main()`. After the fifth iteration, the latch opens, which allows the main thread to resume.

CountDownLatch is a powerful yet easy-to-use synchronization object that is appropriate whenever a thread must wait for one or more events to occur.

CyclicBarrier

A situation not uncommon in concurrent programming occurs when a set of two or more threads must wait at a predetermined execution point until all threads in the set have reached that point. To handle such a situation, the concurrent API supplies the **CyclicBarrier** class. It enables you to define a synchronization object that suspends until the specified number of threads has reached the barrier point.

CyclicBarrier has the following two constructors:

```

CyclicBarrier(int numThreads)
CyclicBarrier(int numThreads, Runnable action)

```

Here, `numThreads` specifies the number of threads that must reach the barrier before execution continues. In the second form, `action` specifies a thread that will be executed when the barrier is reached.

Here is the general procedure that you will follow to use **CyclicBarrier**. First, create a **CyclicBarrier** object, specifying the number of threads that you will be waiting for. Next, when each thread reaches the barrier, have it call `await()` on that object. This will pause execution of the thread until all of the other threads also call `await()`. Once the specified number of threads has reached the barrier, `await()` will return and execution will resume. Also, if you have specified an action, then that thread is executed.

The `await()` method has the following two forms:

```

int await() throws InterruptedException, BrokenBarrierException
int await(long wait, TimeUnit tu)
    throws InterruptedException, BrokenBarrierException, TimeoutException

```

The first form waits until all the threads have reached the barrier point. The second form waits only for the period of time specified by `wait`. The units represented by `wait` are specified by `tu`. Both forms return a value that indicates the order that the threads arrive at the barrier point. The first thread returns a value equal to the number of threads waited upon minus one. The last thread returns zero.

Here is an example that illustrates **CyclicBarrier**. It waits until a set of three threads has reached the barrier. When that occurs, the thread specified by **BarAction** executes.

```
// An example of CyclicBarrier.

import java.util.concurrent.*;

class BarDemo {
    public static void main(String args[]) {
        CyclicBarrier cb = new CyclicBarrier(3, new BarAction() );

        System.out.println("Starting");

        new MyThread(cb, "A");
        new MyThread(cb, "B");
        new MyThread(cb, "C");

    }
}

// A thread of execution that uses a CyclicBarrier.

class MyThread implements Runnable {
    CyclicBarrier cbar;
    String name;

    MyThread(CyclicBarrier c, String n) {
        cbar = c;
        name = n;
        new Thread(this).start();
    }

    public void run() {

        System.out.println(name);

        try {
            cbar.await();
        } catch (BrokenBarrierException exc) {
            System.out.println(exc);
        } catch (InterruptedException exc) {
            System.out.println(exc);
        }
    }
}

// An object of this class is called when the
// CyclicBarrier ends.
class BarAction implements Runnable {
    public void run() {
        System.out.println("Barrier Reached!");
    }
}
```

The output is shown here. (The precise order in which the threads execute may vary.)

```
Starting
A
B
C
Barrier Reached!
```

A **CyclicBarrier** can be reused because it will release waiting threads each time the specified number of threads calls **await()**. For example, if you change **main()** in the preceding program so that it looks like this:

```
public static void main(String args[]) {
    CyclicBarrier cb = new CyclicBarrier(3, new BarAction());
    System.out.println("Starting");
    new MyThread(cb, "A");
    new MyThread(cb, "B");
    new MyThread(cb, "C");
    new MyThread(cb, "X");
    new MyThread(cb, "Y");
    new MyThread(cb, "Z");
}
```

the following output will be produced. (The precise order in which the threads execute may vary.)

```
Starting
A
B
C
Barrier Reached!
X
Y
Z
Barrier Reached!
```

As the preceding example shows, the **CyclicBarrier** offers a streamlined solution to what was previously a complicated problem.

Exchanger

Perhaps the most interesting of the synchronization classes is **Exchanger**. It is designed to simplify the exchange of data between two threads. The operation of an **Exchanger** is astoundingly simple: it simply waits until two separate threads call its **exchange()** method. When that occurs, it exchanges the data supplied by the threads. This mechanism is both elegant and easy to use. Uses for **Exchanger** are easy to imagine. For example, one thread might prepare a buffer for receiving information over a network connection. Another thread might fill that buffer with the information from the connection. The two threads work together so that each time a new buffer is needed, an exchange is made.

Exchanger is a generic class that is declared as shown here:

```
Exchanger<V>
```

Here, **V** specifies the type of the data being exchanged.

The only method defined by **Exchanger** is **exchange()**, which has the two forms shown here:

```
V exchange(V objRef) throws InterruptedException
```

```
V exchange(V objRef, long wait, TimeUnit tu)
throws InterruptedException, TimeoutException
```

Here, *objRef* is a reference to the data to exchange. The data received from the other thread is returned. The second form of **exchange()** allows a time-out period to be specified. The key point about **exchange()** is that it won't succeed until it has been called on the same **Exchanger** object by two separate threads. Thus, **exchange()** synchronizes the exchange of the data.

Here is an example that demonstrates **Exchanger**. It creates two threads. One thread creates an empty buffer that will receive the data put into it by the second thread. In this case, the data is a string. Thus, the first thread exchanges an empty string for a full one.

```
// An example of Exchanger.

import java.util.concurrent.Exchanger;

class ExgrDemo {
    public static void main(String args[]) {
        Exchanger<String> exgr = new Exchanger<String>();

        new UseString(exgr);
        new MakeString(exgr);
    }
}

// A Thread that constructs a string.
class MakeString implements Runnable {
    Exchanger<String> ex;
    String str;

    MakeString(Exchanger<String> c) {
        ex = c;
        str = new String();
        new Thread(this).start();
    }

    public void run() {
        char ch = 'A';

        for(int i = 0; i < 3; i++) {

            // Fill Buffer
            for(int j = 0; j < 5; j++)
                str += ch;
        }
    }
}
```

```

        str += ch++;
    }

    try {
        // Exchange a full buffer for an empty one.
        str = ex.exchange(str);
    } catch(InterruptedException exc) {
        System.out.println(exc);
    }
}

// A Thread that uses a string.
class UseString implements Runnable {
    Exchanger<String> ex;
    String str;
    UseString(Exchanger<String> c) {
        ex = c;
        new Thread(this).start();
    }

    public void run() {

        for(int i=0; i < 3; i++) {
            try {
                // Exchange an empty buffer for a full one.
                str = ex.exchange(new String());
                System.out.println("Got: " + str);
            } catch(InterruptedException exc) {
                System.out.println(exc);
            }
        }
    }
}

```

Here is the output produced by the program:

```

Got: ABCDE
Got: FGHIJ
Got: KLMNO

```

In the program, the `main()` method creates an `Exchanger` for strings. This object is then used to synchronize the exchange of strings between the `MakeString` and `UseString` classes. The `MakeString` class fills a string with data. The `UseString` exchanges an empty string for a full one. It then displays the contents of the newly constructed string. The exchange of empty and full buffers is synchronized by the `exchange()` method, which is called by both classes' `run()` method.

Phaser

Another synchronization class is called **Phaser**. Its primary purpose is to enable the synchronization of threads that represent one or more phases of activity. For example, you might have a set of threads that implement three phases of an order-processing application. In the first phase, separate threads are used to validate customer information, check inventory, and confirm pricing. When that phase is complete, the second phase has two threads that compute shipping costs and all applicable tax. After that, a final phase confirms payment and determines estimated shipping time. In the past, to synchronize the multiple threads that comprise this scenario would require a bit of work on your part. With the inclusion of **Phaser**, the process is now much easier.

To begin, it helps to know that a **Phaser** works a bit like a **CyclicBarrier**, described earlier, except that it supports multiple phases. As a result, **Phaser** lets you define a synchronization object that waits until a specific phase has completed. It then advances to the next phase, again waiting until that phase concludes. It is important to understand that **Phaser** can also be used to synchronize only a single phase. In this regard, it acts much like a **CyclicBarrier**. However, its primary use is to synchronize multiple phases.

Phaser defines four constructors. Here are the two used in this section:

```
Phaser( )
Phaser(int numParties)
```

The first creates a phaser that has a registration count of zero. The second sets the registration count to *numParties*. The term *party* is often applied to the objects that register with a phaser. Although often there is a one-to-correspondence between the number of registrants and the number of threads being synchronized, this is not required. In both cases, the current phase is zero. That is, when a **Phaser** is created, it is initially at phase zero.

In general, here is how you use **Phaser**. First, create a new instance of **Phaser**. Next, register one or more parties with the phaser, either by calling **register()** or by specifying the number of parties in the constructor. For each registered party, have the phaser wait until all registered parties complete a phase. A party signals this by calling one of a variety of methods supplied by **Phaser**, such as **arrive()** or **arriveAndAwaitAdvance()**. After all parties have arrived, the phase is complete, and the phaser can move on to the next phase (if there is one), or terminate. The following sections explain the process in detail.

To register parties after a **Phaser** has been constructed, call **register()**. It is shown here:

```
int register()
```

It returns the phase number of the phase to which it is registered.

To signal that a party has completed a phase, it must call **arrive()** or some variation of **arrive()**. When the number of arrivals equals the number of registered parties, the phase is completed and the **Phaser** moves on to the next phase (if there is one). The **arrive()** method has this general form:

```
int arrive()
```

This method signals that a party (normally a thread of execution) has completed some task (or portion of a task). It returns the current phase number. If the phaser has been terminated, then it returns a negative value. The **arrive()** method does not suspend

execution of the calling thread. This means that it does not wait for the phase to be completed. This method should be called only by a registered party.

If you want to indicate the completion of a phase and then wait until all other registrants have also completed that phase, use **arriveAndAwaitAdvance()**. It is shown here:

```
int arriveAndAwaitAdvance()
```

It waits until all parties have arrived. It returns the next phase number or a negative value if the phaser has been terminated. This method should be called only by a registered party.

A thread can arrive and then deregister itself by calling **arriveAndDeregister()**. It is shown here:

```
int arriveAndDeregister()
```

It returns the current phase number or a negative value if the phaser has been terminated. It does not wait until the phase is complete. This method should be called only by a registered party.

To obtain the current phase number, call **getPhase()**, which is shown here:

```
final int getPhase()
```

When a **Phaser** is created, the first phase will be 0, the second phase 1, the third phase 2, and so on. A negative value is returned if the invoking **Phaser** has been terminated.

Here is an example that shows **Phaser** in action. It creates three threads, each of which have three phases. It uses a **Phaser** to synchronize each phase.

```
// An example of Phaser.

import java.util.concurrent.*;

class PhaserDemo {
    public static void main(String args[]) {
        Phaser phsr = new Phaser(1);
        int curPhase;

        System.out.println("Starting");

        new MyThread(phsr, "A");
        new MyThread(phsr, "B");
        new MyThread(phsr, "C");

        // Wait for all threads to complete phase one.
        curPhase = phsr.getPhase();
        phsr.arriveAndAwaitAdvance();
        System.out.println("Phase " + curPhase + " Complete");

        // Wait for all threads to complete phase two.
        curPhase = phsr.getPhase();
        phsr.arriveAndAwaitAdvance();
        System.out.println("Phase " + curPhase + " Complete");

        curPhase = phsr.getPhase();
        phsr.arriveAndAwaitAdvance();
        System.out.println("Phase " + curPhase + " Complete");
    }
}
```

```

// Deregister the main thread.
phsr.arriveAndDeregister();

if(phsr.isTerminated())
    System.out.println("The Phaser is terminated");
}

}

// A thread of execution that uses a Phaser.
class MyThread implements Runnable {
    Phaser phsr;
    String name;

    MyThread(Phaser p, String n) {
        phsr = p;
        name = n;
        phsr.register();
        new Thread(this).start();
    }

    public void run() {

        System.out.println("Thread " + name + " Beginning Phase One");
        phsr.arriveAndAwaitAdvance(); // Signal arrival.

        // Pause a bit to prevent jumbled output. This is for illustration
        // only. It is not required for the proper operation of the phaser.
        try {
            Thread.sleep(10);
        } catch(InterruptedException e) {
            System.out.println(e);
        }

        System.out.println("Thread " + name + " Beginning Phase Two");
        phsr.arriveAndAwaitAdvance(); // Signal arrival.

        // Pause a bit to prevent jumbled output. This is for illustration
        // only. It is not required for the proper operation of the phaser.
        try {
            Thread.sleep(10);
        } catch(InterruptedException e) {
            System.out.println(e);
        }

        System.out.println("Thread " + name + " Beginning Phase Three");
        phsr.arriveAndDeregister(); // Signal arrival and deregister.
    }
}

```

The output is shown here:

```

Starting
Thread A Beginning Phase One
Thread C Beginning Phase One

```

```

Thread B Beginning Phase One
Phase 0 Complete
Thread B Beginning Phase Two
Thread C Beginning Phase Two
Thread A Beginning Phase Two
Phase 1 Complete
Thread C Beginning Phase Three
Thread B Beginning Phase Three
Thread A Beginning Phase Three
Phase 2 Complete
The Phaser is terminated

```

Let's look closely at the key sections of the program. First, in `main()`, a `Phaser` called `phsr` is created with an initial party count of 1 (which corresponds to the main thread). Then three threads are started by creating three `MyThread` objects. Notice that `MyThread` is passed a reference to `phsr` (the phaser). The `MyThread` objects use this phaser to synchronize their activities. Next, `main()` calls `getPhase()` to obtain the current phase number (which is initially zero) and then calls `arriveAndAwaitAdvance()`. This causes `main()` to suspend until phase zero has completed. This won't happen until all `MyThreads` also call `arriveAndAwaitAdvance()`. When this occurs, `main()` will resume execution, at which point it displays that phase zero has completed, and it moves on to the next phase. This process repeats until all three phases have finished. Then, `main()` calls `arriveAndDeregister()`. At that point, all three `MyThreads` have also deregistered. Since this results in there being no registered parties when the phaser advances to the next phase, the phaser is terminated.

Now look at `MyThread`. First, notice that the constructor is passed a reference to the phaser that it will use and then registers with the new thread as a party on that phaser. Thus, each new `MyThread` becomes a party registered with the passed-in phaser. Also notice that each thread has three phases. In this example, each phase consists of a placeholder that simply displays the name of the thread and what it is doing. Obviously, in real-world code, the thread would be performing more meaningful actions. Between the first two phases, the thread calls `arriveAndAwaitAdvance()`. Thus, each thread waits until all threads have completed the phase (and the main thread is ready). After all threads have arrived (including the main thread), the phaser moves on to the next phase. After the third phase, each thread deregisters itself with a call to `arriveAndDeregister()`. As the comments in `MyThread` explain, the calls to `sleep()` are used for the purposes of illustration to ensure that the output is not jumbled because of the multithreading. They are not needed to make the phaser work properly. If you remove them, the output may look a bit jumbled, but the phases will still be synchronized correctly.

One other point: Although the preceding example used three threads that were all of the same type, this is not a requirement. Each party that uses a phaser can be unique, with each performing some separate task.

It is possible to take control of precisely what happens when a phase advance occurs. To do this, you must override the `onAdvance()` method. This method is called by the run time when a `Phaser` advances from one phase to the next. It is shown here:

```
protected boolean onAdvance(int phase, int numParties)
```

Here, `phase` will contain the current phase number prior to being incremented and `numParties` will contain the number of registered parties. To terminate the phaser, `onAdvance()` must return `true`. To keep the phaser alive, `onAdvance()` must return `false`.

The default version of **onAdvance()** returns **true** (thus terminating the phaser) when there are no registered parties. As a general rule, your override should also follow this practice.

One reason to override **onAdvance()** is to enable a phaser to execute a specific number of phases and then stop. The following example gives you the flavor of this usage. It creates a class called **MyPhaser** that extends **Phaser** so that it will run a specified number of phases. It does this by overriding the **onAdvance()** method. The **MyPhaser** constructor accepts one argument, which specifies the number of phases to execute. Notice that **MyPhaser** automatically registers one party. This behavior is useful in this example, but the needs of your own applications may differ.

```
// Extend Phaser and override onAdvance() so that only a specific
// number of phases are executed.

import java.util.concurrent.*;

// Extend MyPhaser to allow only a specific number of phases
// to be executed.
class MyPhaser extends Phaser {
    int numPhases;

    MyPhaser(int parties, int phaseCount) {
        super(parties);
        numPhases = phaseCount - 1;
    }

    // Override onAdvance() to execute the specified
    // number of phases.
    protected boolean onAdvance(int p, int regParties) {
        // This println() statement is for illustration only.
        // Normally, onAdvance() will not display output.
        System.out.println("Phase " + p + " completed.\n");

        // If all phases have completed, return true
        if(p == numPhases || regParties == 0) return true;

        // Otherwise, return false.
        return false;
    }
}

class PhaserDemo2 {
    public static void main(String args[]) {

        MyPhaser phsr = new MyPhaser(1, 4);

        System.out.println("Starting\n");

        new MyThread(phsr, "A");
        new MyThread(phsr, "B");
        new MyThread(phsr, "C");

        // Wait for the specified number of phases to complete.
        while(!phsr.isTerminated()) {
```

```
    phsr.arriveAndAwaitAdvance() ;  
}  
  
    System.out.println("The Phaser is terminated");  
}  
}  
  
// A thread of execution that uses a Phaser.  
class MyThread implements Runnable {  
    Phaser phsr;  
    String name;  
  
    MyThread(Phaser p, String n) {  
        phsr = p;  
        name = n;  
        phsr.register();  
        new Thread(this).start();  
    }  
  
    public void run() {  
  
        while(!phsr.isTerminated()) {  
            System.out.println("Thread " + name + " Beginning Phase " +  
                phsr.getPhase());  
  
            phsr.arriveAndAwaitAdvance();  
  
            // Pause a bit to prevent jumbled output. This is for illustration  
            // only. It is not required for the proper operation of the phaser.  
            try {  
                Thread.sleep(10);  
            } catch(InterruptedException e) {  
                System.out.println(e);  
            }  
        }  
    }  
}
```

The output from the program is shown here:

```
Starting  
  
Thread B Beginning Phase 0  
Thread A Beginning Phase 0  
Thread C Beginning Phase 0  
Phase 0 completed.  
  
Thread A Beginning Phase 1  
Thread B Beginning Phase 1  
Thread C Beginning Phase 1  
Phase 1 completed.  
  
Thread C Beginning Phase 2  
Thread B Beginning Phase 2  
Thread A Beginning Phase 2  
Phase 2 completed.
```

```

Thread C Beginning Phase 3
Thread B Beginning Phase 3
Thread A Beginning Phase 3
Phase 3 completed.

```

The Phaser is terminated

Inside **main()**, one instance of **Phaser** is created. It is passed 4 as an argument, which means that it will execute four phases and then stop. Next, three threads are created and then the following loop is entered:

```

// Wait for the specified number of phases to complete.
while(!phsr.isTerminated()) {
    phsr.arriveAndAwaitAdvance();
}

```

This loop simply calls **arriveAndAwaitAdvance()** until the phaser is terminated. The phaser won't terminate until the specified number of phases have been executed. In this case, the loop continues to execute until four phases have run. Next, notice that the threads also call **arriveAndAwaitAdvance()** within a loop that runs until the phaser is terminated. This means that they will execute until the specified number of phases has been completed.

Now, look closely at the code for **onAdvance()**. Each time **onAdvance()** is called, it is passed the current phase and the number of registered parties. If the current phase equals the specified phase, or if the number of registered parties is zero, **onAdvance()** returns **true**, thus stopping the phaser. This is accomplished with this line of code:

```

// If all phases have completed, return true
if(p == numPhases || regParties == 0) return true;

```

As you can see, very little code is needed to accommodate the desired outcome.

Before moving on, it is useful to point out that you don't necessarily need to explicitly extend **Phaser** as the previous example does to simply override **onAdvance()**. In some cases, more compact code can be created by using an anonymous inner class to override **onAdvance()**.

Phaser has additional capabilities that may be of use in your applications. You can wait for a specific phase by calling **awaitAdvance()**, which is shown here:

```
int awaitAdvance(int phase)
```

Here, *phase* indicates the phase number on which **awaitAdvance()** will wait until a transition to the next phase takes place. It will return immediately if the argument passed to *phase* is not equal to the current phase. It will also return immediately if the phaser is terminated. However, if *phase* is passed the current phase, then it will wait until the phase increments. This method should be called only by a registered party. There is also an interruptible version of this method called **awaitAdvanceInterruptibly()**.

To register more than one party, call **bulkRegister()**. To obtain the number of registered parties, call **getRegisteredParties()**. You can also obtain the number of arrived parties and unarrived parties by calling **getArrivedParties()** and **getUnarrivedParties()**, respectively. To force the phaser to enter a terminated state, call **forceTermination()**.

Phaser also lets you create a tree of phasers. This is supported by two additional constructors, which let you specify the parent, and the **getParent()** method.

Using an Executor

The concurrent API supplies a feature called an *executor* that initiates and controls the execution of threads. As such, an executor offers an alternative to managing threads through the **Thread** class.

At the core of an executor is the **Executor** interface. It defines the following method:

```
void execute(Runnable thread)
```

The thread specified by *thread* is executed. Thus, **execute()** starts the specified thread.

The **ExecutorService** interface extends **Executor** by adding methods that help manage and control the execution of threads. For example, **ExecutorService** defines **shutdown()**, shown here, which stops the invoking **ExecutorService**.

```
void shutdown()
```

ExecutorService also defines methods that execute threads that return results, that execute a set of threads, and that determine the shutdown status. We will look at several of these methods a little later.

Also defined is the interface **ScheduledExecutorService**, which extends **ExecutorService** to support the scheduling of threads.

The concurrent API defines three predefined executor classes: **ThreadPoolExecutor** and **ScheduledThreadPoolExecutor**, and **ForkJoinPool**. **ThreadPoolExecutor** implements the **Executor** and **ExecutorService** interfaces and provides support for a managed pool of threads. **ScheduledThreadPoolExecutor** also implements the **ScheduledExecutorService** interface to allow a pool of threads to be scheduled. **ForkJoinPool** implements the **Executor** and **ExecutorService** interfaces and is used by the Fork/Join Framework. It is described later in this chapter.

A thread pool provides a set of threads that is used to execute various tasks. Instead of each task using its own thread, the threads in the pool are used. This reduces the overhead associated with creating many separate threads. Although you can use **ThreadPoolExecutor** and **ScheduledThreadPoolExecutor** directly, most often you will want to obtain an executor by calling one of the following static factory methods defined by the **Executors** utility class. Here are some examples:

```
static ExecutorService newCachedThreadPool()
static ExecutorService newFixedThreadPool(int numThreads)
static ScheduledExecutorService newScheduledThreadPool(int numThreads)
```

newCachedThreadPool() creates a thread pool that adds threads as needed but reuses threads if possible. **newFixedThreadPool()** creates a thread pool that consists of a specified number of threads. **newScheduledThreadPool()** creates a thread pool that supports thread scheduling. Each returns a reference to an **ExecutorService** that can be used to manage the pool.

A Simple Executor Example

Before going any further, a simple example that uses an executor will be of value. The following program creates a fixed thread pool that contains two threads. It then uses that

pool to execute four tasks. Thus, four tasks share the two threads that are in the pool. After the tasks finish, the pool is shut down and the program ends.

```
// A simple example that uses an Executor.

import java.util.concurrent.*;

class SimpExec {
    public static void main(String args[]) {
        CountDownLatch cdl = new CountDownLatch(5);
        CountDownLatch cdl2 = new CountDownLatch(5);
        CountDownLatch cdl3 = new CountDownLatch(5);
        CountDownLatch cdl4 = new CountDownLatch(5);
        ExecutorService es = Executors.newFixedThreadPool(2);

        System.out.println("Starting");

        // Start the threads.
        es.execute(new MyThread(cdl, "A"));
        es.execute(new MyThread(cdl2, "B"));
        es.execute(new MyThread(cdl3, "C"));
        es.execute(new MyThread(cdl4, "D"));

        try {
            cdl.await();
            cdl2.await();
            cdl3.await();
            cdl4.await();
        } catch (InterruptedException exc) {
            System.out.println(exc);
        }

        es.shutdown();
        System.out.println("Done");
    }
}

class MyThread implements Runnable {
    String name;
    CountDownLatch latch;

    MyThread(CountDownLatch c, String n) {
        latch = c;
        name = n;

        new Thread(this);
    }

    public void run() {

        for(int i = 0; i < 5; i++) {
            System.out.println(name + ":" + i);
            latch.countDown();
        }
    }
}
```

The output from the program is shown here. (The precise order in which the threads execute may vary.)

```
Starting
A: 0
A: 1
A: 2
A: 3
A: 4
C: 0
C: 1
C: 2
C: 3
C: 4
D: 0
D: 1
D: 2
D: 3
D: 4
B: 0
B: 1
B: 2
B: 3
B: 4
Done
```

As the output shows, even though the thread pool contains only two threads, all four tasks are still executed. However, only two can run at the same time. The others must wait until one of the pooled threads is available for use.

The call to `shutdown()` is important. If it were not present in the program, then the program would not terminate because the executor would remain active. To try this for yourself, simply comment out the call to `shutdown()` and observe the result.

Using Callable and Future

One of the most interesting features of the concurrent API is the **Callable** interface. This interface represents a thread that returns a value. An application can use **Callable** objects to compute results that are then returned to the invoking thread. This is a powerful mechanism because it facilitates the coding of many types of numerical computations in which partial results are computed simultaneously. It can also be used to run a thread that returns a status code that indicates the successful completion of the thread.

Callable is a generic interface that is defined like this:

```
interface Callable<V>
```

Here, **V** indicates the type of data returned by the task. **Callable** defines only one method, `call()`, which is shown here:

```
V call() throws Exception
```

Inside `call()`, you define the task that you want performed. After that task completes, you return the result. If the result cannot be computed, `call()` must throw an exception.

A **Callable** task is executed by an **ExecutorService**, by calling its **submit()** method. There are three forms of **submit()**, but only one is used to execute a **Callable**. It is shown here:

```
<T> Future<T> submit(Callable<T> task)
```

Here, *task* is the **Callable** object that will be executed in its own thread. The result is returned through an object of type **Future**.

Future is a generic interface that represents the value that will be returned by a **Callable** object. Because this value is obtained at some future time, the name **Future** is appropriate. **Future** is defined like this:

```
interface Future<V>
```

Here, **V** specifies the type of the result.

To obtain the returned value, you will call **Future**'s **get()** method, which has these two forms:

```
V get()
    throws InterruptedException, ExecutionException
```

```
V get(long wait, TimeUnit tu)
    throws InterruptedException, ExecutionException, TimeoutException
```

The first form waits for the result indefinitely. The second form allows you to specify a timeout period in *wait*. The units of *wait* are passed in *tu*, which is an object of the **TimeUnit** enumeration, described later in this chapter.

The following program illustrates **Callable** and **Future** by creating three tasks that perform three different computations. The first returns the summation of a value, the second computes the length of the hypotenuse of a right triangle given the length of its sides, and the third computes the factorial of a value. All three computations occur simultaneously.

```
// An example that uses a Callable.

import java.util.concurrent.*;

class CallableDemo {
    public static void main(String args[]) {
        ExecutorService es = Executors.newFixedThreadPool(3);
        Future<Integer> f;
        Future<Double> f2;
        Future<Integer> f3;

        System.out.println("Starting");

        f = es.submit(new Sum(10));
        f2 = es.submit(new Hypot(3, 4));
        f3 = es.submit(new Factorial(5));

        try {
            System.out.println(f.get());
            System.out.println(f2.get());
            System.out.println(f3.get());
        }
    }
}
```

```
        } catch (InterruptedException exc) {
            System.out.println(exc);
        }
        catch (ExecutionException exc) {
            System.out.println(exc);
        }

        es.shutdown();
        System.out.println("Done");
    }
}

// Following are three computational threads.

class Sum implements Callable<Integer> {
    int stop;

    Sum(int v) { stop = v; }

    public Integer call() {
        int sum = 0;
        for(int i = 1; i <= stop; i++) {
            sum += i;
        }
        return sum;
    }
}

class Hypot implements Callable<Double> {
    double side1, side2;

    Hypot(double s1, double s2) {
        side1 = s1;
        side2 = s2;
    }

    public Double call() {
        return Math.sqrt((side1*side1) + (side2*side2));
    }
}

class Factorial implements Callable<Integer> {
    int stop;

    Factorial(int v) { stop = v; }

    public Integer call() {
        int fact = 1;
        for(int i = 2; i <= stop; i++) {
            fact *= i;
        }
        return fact;
    }
}
```

The output is shown here:

```
Starting
55
5.0
120
Done
```

The TimeUnit Enumeration

The concurrent API defines several methods that take an argument of type **TimeUnit**, which indicates a time-out period. **TimeUnit** is an enumeration that is used to specify the *granularity* (or resolution) of the timing. **TimeUnit** is defined within **java.util.concurrent**. It can be one of the following values:

```
DAYS
HOURS
MINUTES
SECONDS
MICROSECONDS
MILLISECONDS
NANOSECONDS
```

Although **TimeUnit** lets you specify any of these values in calls to methods that take a timing argument, there is no guarantee that the system is capable of the specified resolution.

Here is an example that uses **TimeUnit**. The **CallableDemo** class, shown in the previous section, is modified as shown next to use the second form of **get()** that takes a **TimeUnit** argument.

```
try {
    System.out.println(f.get(10, TimeUnit.MILLISECONDS));
    System.out.println(f2.get(10, TimeUnit.MILLISECONDS));
    System.out.println(f3.get(10, TimeUnit.MILLISECONDS));
} catch (InterruptedException exc) {
    System.out.println(exc);
}
catch (ExecutionException exc) {
    System.out.println(exc);
} catch (TimeoutException exc) {
    System.out.println(exc);
}
```

In this version, no call to **get()** will wait more than 10 milliseconds.

The **TimeUnit** enumeration defines various methods that convert between units. These are shown here:

```
long convert(long tval, TimeUnit tu)
long toMicros(long tval)
long toMillis(long tval)
long toNanos(long tval)
```

```
long toSeconds(long tval)
long toDays(long tval)
long toHours(long tval)
long toMinutes(long tval)
```

The **convert()** method converts *tval* into the specified unit and returns the result. The **to** methods perform the indicated conversion and return the result.

TimeUnit also defines the following timing methods:

```
void sleep(long delay) throws InterruptedException
void timedJoin(Thread thrd, long delay) throws InterruptedException
void timedWait(Object obj, long delay) throws InterruptedException
```

Here, **sleep()** pauses execution for the specified delay period, which is specified in terms of the invoking enumeration constant. It translates into a call to **Thread.sleep()**. The **timedJoin()** method is a specialized version of **Thread.join()** in which *thrd* pauses for the time period specified by *delay*, which is described in terms of the invoking time unit. The **timedWait()** method is a specialized version of **Object.wait()** in which *obj* is waited on for the period of time specified by *delay*, which is described in terms of the invoking time unit.

The Concurrent Collections

As explained, the concurrent API defines several collection classes that have been engineered for concurrent operation. They include:

- ArrayBlockingQueue
- ConcurrentHashMap
- ConcurrentLinkedDeque
- ConcurrentLinkedQueue
- ConcurrentSkipListMap
- ConcurrentSkipListSet
- CopyOnWriteArrayList
- CopyOnWriteArraySet
- DelayQueue
- LinkedBlockingDeque
- LinkedBlockingQueue
- LinkedTransferQueue
- PriorityBlockingQueue
- SynchronousQueue

These offer concurrent alternatives to their related classes defined by the Collections Framework. These collections work much like the other collections except that they provide concurrency support. Programmers familiar with the Collections Framework will have no trouble using these concurrent collections.

Locks

The **java.util.concurrent.locks** package provides support for *locks*, which are objects that offer an alternative to using **synchronized** to control access to a shared resource. In general, here is how a lock works. Before accessing a shared resource, the lock that protects that

resource is acquired. When access to the resource is complete, the lock is released. If a second thread attempts to acquire the lock when it is in use by another thread, the second thread will suspend until the lock is released. In this way, conflicting access to a shared resource is prevented.

Locks are particularly useful when multiple threads need to access the value of shared data. For example, an inventory application might have a thread that first confirms that an item is in stock and then decreases the number of items on hand as each sale occurs. If two or more of these threads are running, then without some form of synchronization, it would be possible for one thread to be in the middle of a transaction when the second thread begins its transaction. The result could be that both threads would assume that adequate inventory exists, even if there is only sufficient inventory on hand to satisfy one sale. In this type of situation, a lock offers a convenient means of handling the needed synchronization.

The **Lock** interface defines a lock. The methods defined by **Lock** are shown in Table 28-1. In general, to acquire a lock, call **lock()**. If the lock is unavailable, **lock()** will wait. To release a lock, call **unlock()**. To see if a lock is available, and to acquire it if it is, call **tryLock()**. This method will not wait for the lock if it is unavailable. Instead, it returns **true** if the lock is acquired and **false** otherwise. The **newCondition()** method returns a **Condition** object associated with the lock. Using a **Condition**, you gain detailed control of the lock through methods such as **await()** and **signal()**, which provide functionality similar to **Object.wait()** and **Object.notify()**.

java.util.concurrent.locks supplies an implementation of **Lock** called **ReentrantLock**. **ReentrantLock** implements a *reentrant lock*, which is a lock that can be repeatedly entered by the thread that currently holds the lock. Of course, in the case of a thread reentering a lock, all calls to **lock()** must be offset by an equal number of calls to **unlock()**. Otherwise, a thread seeking to acquire the lock will suspend until the lock is not in use.

Method	Description
<code>void lock()</code>	Waits until the invoking lock can be acquired.
<code>void lockInterruptibly()</code> throws <code>InterruptedException</code>	Waits until the invoking lock can be acquired, unless interrupted.
<code>Condition newCondition()</code>	Returns a Condition object that is associated with the invoking lock.
<code>boolean tryLock()</code>	Attempts to acquire the lock. This method will not wait if the lock is unavailable. Instead, it returns true if the lock has been acquired and false if the lock is currently in use by another thread.
<code>boolean tryLock(long wait, TimeUnit tu)</code> throws <code>InterruptedException</code>	Attempts to acquire the lock. If the lock is unavailable, this method will wait no longer than the period specified by <i>wait</i> , which is in <i>tu</i> units. It returns true if the lock has been acquired and false if the lock cannot be acquired within the specified period.
<code>void unlock()</code>	Releases the lock.

Table 28-1 The **Lock** Methods

The following program demonstrates the use of a lock. It creates two threads that access a shared resource called **Shared.count**. Before a thread can access **Shared.count**, it must obtain a lock. After obtaining the lock, **Shared.count** is incremented and then, before releasing the lock, the thread sleeps. This causes the second thread to attempt to obtain the lock. However, because the lock is still held by the first thread, the second thread must wait until the first thread stops sleeping and releases the lock. The output shows that access to **Shared.count** is, indeed, synchronized by the lock.

```
// A simple lock example.

import java.util.concurrent.locks.*;

class LockDemo {

    public static void main(String args[]) {
        ReentrantLock lock = new ReentrantLock();

        new LockThread(lock, "A");
        new LockThread(lock, "B");
    }
}

// A shared resource.
class Shared {
    static int count = 0;
}

// A thread of execution that increments count.
class LockThread implements Runnable {
    String name;
    ReentrantLock lock;

    LockThread(ReentrantLock lk, String n) {
        lock = lk;
        name = n;
        new Thread(this).start();
    }

    public void run() {

        System.out.println("Starting " + name);

        try {
            // First, lock count.
            System.out.println(name + " is waiting to lock count.");
            lock.lock();
            System.out.println(name + " is locking count.");

            Shared.count++;
            System.out.println(name + ": " + Shared.count);

            // Now, allow a context switch -- if possible.
            System.out.println(name + " is sleeping.");
        }
    }
}
```

```
    Thread.sleep(1000);
} catch (InterruptedException exc) {
    System.out.println(exc);
} finally {
    // Unlock
    System.out.println(name + " is unlocking count.");
    lock.unlock();
}
}
```

The output is shown here. (The precise order in which the threads execute may vary.)

```
Starting A
A is waiting to lock count.
A is locking count.
A: 1
A is sleeping.
Starting B
B is waiting to lock count.
A is unlocking count.
B is locking count.
B: 2
B is sleeping.
B is unlocking count.
```

`java.util.concurrent.locks` also defines the **ReadWriteLock** interface. This interface specifies a lock that maintains separate locks for read and write access. This enables multiple locks to be granted for readers of a resource as long as the resource is not being written. **ReentrantReadWriteLock** provides an implementation of **ReadWriteLock**.

NOTE JDK 8 adds a specialized lock called **StampedLock**. It does not implement the **Lock** or **ReadWriteLock** interfaces. It does, however, provide a mechanism that enables aspects of it to be used like a **Lock** or **ReadWriteLock**.

Atomic Operations

java.util.concurrent.atomic offers an alternative to the other synchronization features when reading or writing the value of some types of variables. This package offers methods that get, set, or compare the value of a variable in one uninterruptible (that is, atomic) operation. This means that no lock or other synchronization mechanism is required.

Atomic operations are accomplished through the use of classes, such as **AtomicInteger** and **AtomicLong**, and methods such as **get()**, **set()**, **compareAndSet()**, **decrementAndGet()**, and **getAndSet()**, which perform the action indicated by their names.

Here is an example that demonstrates how access to a shared integer can be synchronized by the use of **AtomicInteger**:

```
// A simple example of Atomic.  
  
import java.util.concurrent.atomic.*;
```

```

class AtomicDemo {
    public static void main(String args[]) {
        new AtomThread("A");
        new AtomThread("B");
        new AtomThread("C");
    }
}

class Shared {
    static AtomicInteger ai = new AtomicInteger(0);
}

// A thread of execution that increments count.
class AtomThread implements Runnable {
    String name;

    AtomThread(String n) {
        name = n;
        new Thread(this).start();
    }

    public void run() {
        System.out.println("Starting " + name);

        for(int i=1; i <= 3; i++)
            System.out.println(name + " got: " +
                Shared.ai.getAndSet(i));
    }
}

```

In the program, a static **AtomicInteger** named **ai** is created by **Shared**. Then, three threads of type **AtomThread** are created. Inside **run()**, **Shared.ai** is modified by calling **getAndSet()**. This method returns the previous value and then sets the value to the one passed as an argument. The use of **AtomicInteger** prevents two threads from writing to **ai** at the same time.

In general, the atomic operations offer a convenient (and possibly more efficient) alternative to the other synchronization mechanisms when only a single variable is involved. Beginning with JDK 8, **java.util.concurrent.atomic** also provides four classes that support lock-free cumulative operations. These are **DoubleAccumulator**, **DoubleAdder**, **LongAccumulator**, and **LongAdder**. The accumulator classes support a series of user-specified operations. The adder classes maintain a cumulative sum.

Parallel Programming via the Fork/Join Framework

In recent years, an important new trend has emerged in software development: *parallel programming*. Parallel programming is the name commonly given to the techniques that take advantage of computers that contain two or more processors (multicore). As most readers will know, multicore computers are becoming commonplace. The advantage that multi-processor environments offer is the ability to significantly increase program performance. As a result, there has been a growing need for a mechanism that gives Java

programmers a simple, yet effective way to make use of multiple processors in a clean, scalable manner. To answer this need, JDK 7 added several new classes and interfaces that support parallel programming. They are commonly referred to as the *Fork/Join Framework*. It is one of the more important additions that has recently been made to the Java class library. The Fork/Join Framework is defined in the `java.util.concurrent` package.

The Fork/Join Framework enhances multithreaded programming in two important ways. First, it simplifies the creation and use of multiple threads. Second, it automatically makes use of multiple processors. In other words, by using the Fork/Join Framework you enable your applications to automatically scale to make use of the number of available processors. These two features make the Fork/Join Framework the recommended approach to multithreading when parallel processing is desired.

Before continuing, it is important to point out the distinction between traditional multithreading and parallel programming. In the past, most computers had a single CPU and multithreading was primarily used to take advantage of idle time, such as when a program is waiting for user input. Using this approach, one thread can execute while another is waiting. In other words, on a single-CPU system, multithreading is used to allow two or more tasks to share the CPU. This type of multithreading is typically supported by an object of type **Thread** (as described in Chapter 11). Although this type of multithreading will always remain quite useful, it was not optimized for situations in which two or more CPUs are available (multicore computers).

When multiple CPUs are present, a second type of multithreading capability that supports true parallel execution is required. With two or more CPUs, it is possible to execute portions of a program simultaneously, with each part executing on its own CPU. This can be used to significantly speed up the execution of some types of operations, such as sorting, transforming, or searching a large array. In many cases, these types of operations can be broken down into smaller pieces (each acting on a portion of the array), and each piece can be run on its own CPU. As you can imagine, the gain in efficiency can be enormous. Simply put: Parallel programming will be part of nearly every programmer's future because it offers a way to dramatically improve program performance.

The Main Fork/Join Classes

The Fork/Join Framework is packaged in `java.util.concurrent`. At the core of the Fork/Join Framework are the following four classes:

<code>ForkJoinTask<V></code>	An abstract class that defines a task
<code>ForkJoinPool</code>	Manages the execution of <code>ForkJoinTasks</code>
<code>RecursiveAction</code>	A subclass of <code>ForkJoinTask<V></code> for tasks that do not return values
<code>RecursiveTask<V></code>	A subclass of <code>ForkJoinTask<V></code> for tasks that return values

Here is how they relate. A **ForkJoinPool** manages the execution of **ForkJoinTasks**. **ForkJoinTask** is an abstract class that is extended by the abstract classes **RecursiveAction** and **RecursiveTask**. Typically, your code will extend these classes to create a task. Before looking at the process in detail, an overview of the key aspects of each class will be helpful.

NOTE The class **CountedCompleter** (added by JDK 8) also extends **ForkJoinTask**. However, a discussion of **CountedCompleter** is beyond the scope of this book.

ForkJoinTask<V>

ForkJoinTask<V> is an abstract class that defines a task that can be managed by a **ForkJoinPool**. The type parameter **V** specifies the result type of the task. **ForkJoinTask** differs from **Thread** in that **ForkJoinTask** represents lightweight abstraction of a task, rather than a thread of execution. **ForkJoinTasks** are executed by threads managed by a thread pool of type **ForkJoinPool**. This mechanism allows a large number of tasks to be managed by a small number of actual threads. Thus, **ForkJoinTasks** are very efficient when compared to threads.

ForkJoinTask defines many methods. At the core are **fork()** and **join()**, shown here:

```
final ForkJoinTask<V> fork()
final V join()
```

The **fork()** method submits the invoking task for asynchronous execution of the invoking task. This means that the thread that calls **fork()** continues to run. The **fork()** method returns **this** after the task is scheduled for execution. Prior to JDK 8, **fork()** could be executed only from within the computational portion of another **ForkJoinTask**, which is running within a **ForkJoinPool**. (You will see how to create the computational portion of a task shortly.) However, with the advent of JDK 8, if **fork()** is not called while executing within a **ForkJoinPool**, then a common pool is automatically used. The **join()** method waits until the task on which it is called terminates. The result of the task is returned. Thus, through the use of **fork()** and **join()**, you can start one or more new tasks and then wait for them to finish.

Another important **ForkJoinTask** method is **invoke()**. It combines the fork and join operations into a single call because it begins a task and then waits for it to end. It is shown here:

```
final V invoke()
```

The result of the invoking task is returned.

You can invoke more than one task at a time by using **invokeAll()**. Two of its forms are shown here:

```
static void invokeAll(ForkJoinTask<?> taskA, ForkJoinTask<?> taskB)
static void invokeAll(ForkJoinTask<?> ... taskList)
```

In the first case, *taskA* and *taskB* are executed. In the second case, all specified tasks are executed. In both cases, the calling thread waits until all of the specified tasks have terminated. Prior to JDK 8, the **invokeAll()** method could be executed only from within the computational portion of another **ForkJoinTask**, which is running within a **ForkJoinPool**. JDK 8's inclusion of the common pool relaxed this requirement.

RecursiveAction

A subclass of **ForkJoinTask** is **RecursiveAction**. This class encapsulates a task that does not return a result. Typically, your code will extend **RecursiveAction** to create a task that has a **void** return type. **RecursiveAction** specifies four methods, but only one is usually of interest: the abstract method called **compute()**. When you extend **RecursiveAction** to create a concrete class, you will put the code that defines the task inside **compute()**. The **compute()** method represents the *computational* portion of the task.

The `compute()` method is defined by **RecursiveAction** like this:

```
protected abstract void compute( )
```

Notice that `compute()` is **protected** and **abstract**. This means that it must be implemented by a subclass (unless that subclass is also abstract).

In general, **RecursiveAction** is used to implement a recursive, divide-and-conquer strategy for tasks that don't return results. (See "The Divide-and-Conquer Strategy" later in this chapter.)

RecursiveTask<V>

Another subclass of **ForkJoinTask** is **RecursiveTask<V>**. This class encapsulates a task that returns a result. The result type is specified by **V**. Typically, your code will extend **RecursiveTask<V>** to create a task that returns a value. Like **RecursiveAction**, it too specifies four methods, but often only the abstract `compute()` method is used, which represents the computational portion of the task. When you extend **RecursiveTask<V>** to create a concrete class, put the code that represents the task inside `compute()`. This code must also return the result of the task.

The `compute()` method is defined by **RecursiveTask<V>** like this:

```
protected abstract V compute( )
```

Notice that `compute()` is **protected** and **abstract**. This means that it must be implemented by a subclass. When implemented, it must return the result of the task.

In general, **RecursiveTask** is used to implement a recursive, divide-and-conquer strategy for tasks that return results. (See "The Divide-and-Conquer Strategy" later in this chapter.)

ForkJoinPool

The execution of **ForkJoinTasks** takes place within a **ForkJoinPool**, which also manages the execution of the tasks. Therefore, in order to execute a **ForkJoinTask**, you must first have a **ForkJoinPool**. Beginning with JDK 8, there are two ways to acquire a **ForkJoinPool**. First, you can explicitly create one by using a **ForkJoinPool** constructor. Second, you can use what is referred to as the *common pool*. The common pool (which was added by JDK 8) is a static **ForkJoinPool** that is automatically available for your use. Each method is introduced here, beginning with manually constructing a pool.

ForkJoinPool defines several constructors. Here are two commonly used ones:

`ForkJoinPool()`

`ForkJoinPool(int pLevel)`

The first creates a default pool that supports a level of parallelism equal to the number of processors available in the system. The second lets you specify the level of parallelism. Its value must be greater than zero and not more than the limits of the implementation. The level of parallelism determines the number of threads that can execute concurrently. As a result, the level of parallelism effectively determines the number of tasks that can be executed simultaneously. (Of course, the number of tasks that can execute simultaneously cannot exceed the number of processors.) It is important to understand that the level of parallelism *does not*, however, limit the number of tasks that can be managed by the pool. A **ForkJoinPool** can manage many more tasks than its level of parallelism. Also, the level of parallelism is only a target. It is not a guarantee.

After you have created an instance of **ForkJoinPool**, you can start a task in a number of different ways. The first task started is often thought of as the main task. Frequently, the main task begins subtasks that are also managed by the pool. One common way to begin a main task is to call **invoke()** on the **ForkJoinPool**. It is shown here:

```
<T> T invoke(ForkJoinTask<T> task)
```

This method begins the task specified by *task*, and it returns the result of the task. This means that the calling code waits until **invoke()** returns.

To start a task without waiting for its completion, you can use **execute()**. Here is one of its forms:

```
void execute(ForkJoinTask<?> task)
```

In this case, *task* is started, but the calling code does not wait for its completion. Rather, the calling code continues execution asynchronously.

Beginning with JDK 8, it is not necessary to explicitly construct a **ForkJoinPool** because a common pool is available for your use. In general, if you are not using a pool that you explicitly created, then the common pool will automatically be used. Although it won't always be necessary, you can obtain a reference to the common pool by calling **commonPool()**, which is defined by **ForkJoinPool**. It is shown here:

```
static ForkJoinPool commonPool()
```

A reference to the common pool is returned. The common pool provides a default level of parallelism. It can be set by use of a system property. (See the API documentation for details.) Typically, the default common pool is a good choice for many applications. Of course, you can always construct your own pool.

There are two basic ways to start a task using the common pool. First, you can obtain a reference to the pool by calling **commonPool()** and then use that reference to call **invoke()** or **execute()**, as just described. Second, you can call **ForkJoinTask** methods such as **fork()** or **invoke()** on the task from outside its computational portion. In this case, the common pool will automatically be used. In other words, **fork()** and **invoke()** will start a task using the common pool if the task is not already running within a **ForkJoinPool**.

ForkJoinPool manages the execution of its threads using an approach called *work-stealing*. Each worker thread maintains a queue of tasks. If one worker thread's queue is empty, it will take a task from another worker thread. This adds to overall efficiency and helps maintain a balanced load. (Because of demands on CPU time by other processes in the system, even two worker threads with identical tasks in their respective queues may not complete at the same time.)

One other point: **ForkJoinPool** uses daemon threads. A daemon thread is automatically terminated when all user threads have terminated. Thus, there is no need to explicitly shut down a **ForkJoinPool**. However, with the exception of the common pool, it is possible to do so by calling **shutdown()**. The **shutdown()** method has no effect on the common pool.

The Divide-and-Conquer Strategy

As a general rule, users of the Fork/Join Framework will employ a *divide-and-conquer* strategy that is based on recursion. This is why the two subclasses of **ForkJoinTask** are called **RecursiveAction** and **RecursiveTask**. It is anticipated that you will extend one of these classes when creating your own fork/join task.

The divide-and-conquer strategy is based on recursively dividing a task into smaller subtasks until the size of a subtask is small enough to be handled sequentially. For example, a task that applies a transform to each element in an array of N integers can be broken down into two subtasks in which each transforms half the elements in the array. That is, one subtask transforms the elements 0 to $N/2$, and the other transforms the elements $N/2$ to N . In turn, each subtask can be reduced to another set of subtasks, each transforming half of the remaining elements. This process of dividing the array will continue until a threshold is reached in which a sequential solution is faster than creating another division.

The advantage of the divide-and-conquer strategy is that the processing can occur in parallel. Therefore, instead of cycling through an entire array using a single thread, pieces of the array can be processed simultaneously. Of course, the divide-and-conquer approach works in many cases in which an array (or collection) is not present, but the most common uses involve some type of array, collection, or grouping of data.

One of the keys to best employing the divide-and-conquer strategy is correctly selecting the threshold at which sequential processing (rather than further division) is used. Typically, an optimal threshold is obtained through profiling the execution characteristics. However, very significant speed-ups will still occur even when a less-than-optimal threshold is used. It is, however, best to avoid overly large or overly small thresholds. At the time of this writing, the Java API documentation for **ForkJoinTask<T>** states that, as a rule-of-thumb, a task should perform somewhere between 100 and 10,000 computational steps.

It is also important to understand that the optimal threshold value is also affected by how much time the computation takes. If each computational step is fairly long, then smaller thresholds might be better. Conversely, if each computational step is quite short, then larger thresholds could yield better results. For applications that are to be run on a known system, with a known number of processors, you can use the number of processors to make informed decisions about the threshold value. However, for applications that will be running on a variety of systems, the capabilities of which are not known in advance, you can make no assumptions about the execution environment.

One other point: Although multiple processors may be available on a system, other tasks (and the operating system, itself) will be competing with your application for CPU time. Thus, it is important not to assume that your program will have unrestricted access to all CPUs. Furthermore, different runs of the same program may display different run time characteristics because of varying task loads.

A Simple First Fork/Join Example

At this point, a simple example that demonstrates the Fork/Join Framework and the divide-and-conquer strategy will be helpful. Following is a program that transforms the elements in an array of **double** into their square roots. It does so via a subclass of **RecursiveAction**. Notice that it creates its own **ForkJoinPool**.

```
// A simple example of the basic divide-and-conquer strategy.
// In this case, RecursiveAction is used.
import java.util.concurrent.*;
import java.util.*;

// A ForkJoinTask (via RecursiveAction) that transforms
// the elements in an array of doubles into their square roots.
```

```
class SqrtTransform extends RecursiveAction {
    // The threshold value is arbitrarily set at 1,000 in this example.
    // In real-world code, its optimal value can be determined by
    // profiling and experimentation.
    final int seqThreshold = 1000;

    // Array to be accessed.
    double[] data;

    // Determines what part of data to process.
    int start, end;

    SqrtTransform(double[] vals, int s, int e) {
        data = vals;
        start = s;
        end = e;
    }

    // This is the method in which parallel computation will occur.
    protected void compute() {

        // If number of elements is below the sequential threshold,
        // then process sequentially.
        if((end - start) < seqThreshold) {
            // Transform each element into its square root.
            for(int i = start; i < end; i++) {
                data[i] = Math.sqrt(data[i]);
            }
        }
        else {
            // Otherwise, continue to break the data into smaller pieces.

            // Find the midpoint.
            int middle = (start + end) / 2;

            // Invoke new tasks, using the subdivided data.
            invokeAll(new SqrtTransform(data, start, middle),
                      new SqrtTransform(data, middle, end));
        }
    }
}

// Demonstrate parallel execution.
class ForkJoinDemo {
    public static void main(String args[]) {
        // Create a task pool.
        ForkJoinPool fjp = new ForkJoinPool();

        double[] nums = new double[100000];

        // Give nums some values.
        for(int i = 0; i < nums.length; i++)
            nums[i] = (double) i;
```

```

        System.out.println("A portion of the original sequence:");
        for(int i=0; i < 10; i++)
            System.out.print(nums[i] + " ");
        System.out.println("\n");

        SqrtTransform task = new SqrtTransform(nums, 0, nums.length);

        // Start the main ForkJoinTask.
        fjp.invoke(task);

        System.out.println("A portion of the transformed sequence" +
                           " (to four decimal places):");
        for(int i=0; i < 10; i++)
            System.out.format("%.4f ", nums[i]);
        System.out.println();
    }
}

```

The output from the program is shown here:

```

A portion of the original sequence:
0.0 1.0 2.0 3.0 4.0 5.0 6.0 7.0 8.0 9.0

A portion of the transformed sequence (to four decimal places):
0.0000 1.0000 1.4142 1.7321 2.0000 2.2361 2.4495 2.6458 2.8284 3.0000

```

As you can see, the values of the array elements have been transformed into their square roots.

Let's look closely at how this program works. First, notice that **SqrtTransform** is a class that extends **RecursiveAction**. As explained, **RecursiveAction** extends **ForkJoinTask** for tasks that do not return results. Next, notice the **final** variable **seqThreshold**. This is the value that determines when sequential processing will take place. This value is set (somewhat arbitrarily) to 1,000. Next, notice that a reference to the array to be processed is stored in **data** and that the fields **start** and **end** are used to indicate the boundaries of the elements to be accessed.

The main action of the program takes place in **compute()**. It begins by checking if the number of elements to be processed is below the sequential processing threshold. If it is, then those elements are processed (by computing their square root in this example). If the sequential processing threshold has not been reached, then two new tasks are started by calling **invokeAll()**. In this case, each subtask processes half the elements. As explained earlier, **invokeAll()** waits until both tasks return. After all of the recursive calls unwind, each element in the array will have been modified, with much of the action taking place in parallel (if multiple processors are available).

As mentioned, beginning with JDK 8, it is not necessary to explicitly construct a **ForkJoinPool** because a common pool is available for your use. Furthermore, using the common pool is a simple matter. For example, you can obtain a reference to the common pool by calling the static **commonPool()** method defined by **ForkJoinPool**. Therefore, the preceding program could be rewritten to use the common pool by replacing the call to the **ForkJoinPool** constructor with a call to **commonPool()**, as shown here:

```
ForkJoinPool fjp = ForkJoinPool.commonPool();
```

Alternatively, there is no need to explicitly obtain a reference to the common pool because calling the **ForkJoinTask** methods **invoke()** or **fork()** on a task that is not already part of a pool will cause it to execute within the common pool automatically. For example, in the preceding program, you can eliminate the **fjp** variable entirely and start the task using this line:

```
task.invoke();
```

As this discussion shows, the common pool is one of the enhancements JDK 8 made to the Fork/Join Framework that improves its ease-of-use. Furthermore, in many cases, the common pool is the preferable approach, assuming that JDK 7 compatibility is not required.

Understanding the Impact of the Level of Parallelism

Before moving on, it is important to understand the impact that the level of parallelism has on the performance of a fork/join task and how the parallelism and the threshold interact. The program shown in this section lets you experiment with different degrees of parallelism and threshold values. Assuming that you are using a multicore computer, you can interactively observe the effect of these values.

In the preceding example, the default level of parallelism was used. However, you can specify the level of parallelism that you want. One way is to specify it when you create a **ForkJoinPool** using this constructor:

```
ForkJoinPool(int pLevel)
```

Here, *pLevel* specifies the level of parallelism, which must be greater than zero and less than the implementation defined limit.

The following program creates a fork/join task that transforms an array of **doubles**. The transformation is arbitrary, but it is designed to consume several CPU cycles. This was done to ensure that the effects of changing the threshold or the level of parallelism would be more clearly displayed. To use the program, specify the threshold value and the level of parallelism on the command line. The program then runs the tasks. It also displays the amount of time it takes the tasks to run. To do this, it uses **System.nanoTime()**, which returns the value of the JVM's high-resolution timer.

```
// A simple program that lets you experiment with the effects of
// changing the threshold and parallelism of a ForkJoinTask.
import java.util.concurrent.*;

// A ForkJoinTask (via RecursiveAction) that performs a
// a transform on the elements of an array of doubles.
class Transform extends RecursiveAction {

    // Sequential threshold, which is set by the constructor.
    int seqThreshold;

    // Array to be accessed.
    double[] data;

    // Determines what part of data to process.
    int start, end;
```

```
Transform(double[] vals, int s, int e, int t) {
    data = vals;
    start = s;
    end = e;
    seqThreshold = t;
}

// This is the method in which parallel computation will occur.
protected void compute() {

    // If number of elements is below the sequential threshold,
    // then process sequentially.
    if((end - start) < seqThreshold) {
        // The following code assigns an element at an even index the
        // square root of its original value. An element at an odd
        // index is assigned its cube root. This code is designed
        // to simply consume CPU time so that the effects of concurrent
        // execution are more readily observable.
        for(int i = start; i < end; i++) {
            if((data[i] % 2) == 0)
                data[i] = Math.sqrt(data[i]);
            else
                data[i] = Math.cbrt(data[i]);
        }
    }
    else {
        // Otherwise, continue to break the data into smaller pieces.

        // Find the midpoint.
        int middle = (start + end) / 2;

        // Invoke new tasks, using the subdivided data.
        invokeAll(new Transform(data, start, middle, seqThreshold),
                  new Transform(data, middle, end, seqThreshold));
    }
}

// Demonstrate parallel execution.
class FJExperiment {

    public static void main(String args[]) {
        int pLevel;
        int threshold;

        if(args.length != 2) {
            System.out.println("Usage: FJExperiment parallelism threshold ");
            return;
        }

        pLevel = Integer.parseInt(args[0]);
        threshold = Integer.parseInt(args[1]);

        // These variables are used to time the task.
        long beginT, endT;
```

```
// Create a task pool. Notice that the parallelism level is set.  
ForkJoinPool fjp = new ForkJoinPool(pLevel);  
  
double[] nums = new double[1000000];  
  
for(int i = 0; i < nums.length; i++)  
    nums[i] = (double) i;  
  
Transform task = new Transform(nums, 0, nums.length, threshold);  
  
// Starting timing.  
beginT = System.nanoTime();  
  
// Start the main ForkJoinTask.  
fjp.invoke(task);  
  
// End timing.  
endT = System.nanoTime();  
  
System.out.println("Level of parallelism: " + pLevel);  
System.out.println("Sequential threshold: " + threshold);  
System.out.println("Elapsed time: " + (endT - beginT) + " ns");  
System.out.println();  
}  
}
```

To use the program, specify the level of parallelism followed by the threshold limit. You should try experimenting with different values for each, observing the results. Remember, to be effective, you must run the code on a computer with at least two processors. Also, understand that two different runs may (almost certainly will) produce different results because of the effect of other processes in the system consuming CPU time.

To give you an idea of the difference that parallelism makes, try this experiment. First, execute the program like this:

```
java FJExperiment 1 1000
```

This requests 1 level of parallelism (essentially sequential execution) with a threshold of 1,000. Here is a sample run produced on a dual-core computer:

```
Level of parallelism: 1  
Sequential threshold: 1000  
Elapsed time: 259677487 ns
```

Now, specify 2 levels of parallelism like this:

```
java FJExperiment 2 1000
```

Here is sample output from this run produced by the same dual-core computer:

```
Level of parallelism: 2  
Sequential threshold: 1000  
Elapsed time: 169254472 ns
```

As is evident, adding parallelism substantially decreases execution time, thus increasing the speed of the program. You should experiment with varying the threshold and parallelism on your own computer. The results may surprise you.

Here are two other methods that you might find useful when experimenting with the execution characteristics of a fork/join program. First, you can obtain the level of parallelism by calling `getParallelism()`, which is defined by `ForkJoinPool`. It is shown here:

```
int getParallelism()
```

It returns the parallelism level currently in effect. Recall that for pools that you create, by default, this value will equal the number of available processors. (To obtain the parallelism level for the common pool, you can also use `getCommonPoolParallelism()`, which was added by JDK 8.) Second, you can obtain the number of processors available in the system by calling `availableProcessors()`, which is defined by the `Runtime` class. It is shown here:

```
int availableProcessors()
```

The value returned may change from one call to the next because of other system demands.

An Example that Uses `RecursiveTask<V>`

The two preceding examples are based on `RecursiveAction`, which means that they concurrently execute tasks that do not return results. To create a task that returns a result, use `RecursiveTask`. In general, solutions are designed in the same manner as just shown. The key difference is that the `compute()` method returns a result. Thus, you must aggregate the results, so that when the first invocation finishes, it returns the overall result. Another difference is that you will typically start a subtask by calling `fork()` and `join()` explicitly (rather than implicitly by calling `invokeAll()`, for example).

The following program demonstrates `RecursiveTask`. It creates a task called `Sum` that returns the summation of the values in an array of `double`. In this example, the array consists of 5,000 elements. However, every other value is negative. Thus, the first values in the array are 0, -1, 2, -3, 4, and so on. (So that this example will work with both JDK 7 and JDK 8, it creates its own pool. You might try changing it to use the common pool as an exercise.)

```
// A simple example that uses RecursiveTask<V>.
import java.util.concurrent.*;

// A RecursiveTask that computes the summation of an array of doubles.
class Sum extends RecursiveTask<Double> {

    // The sequential threshold value.
    final int seqThresHold = 500;

    // Array to be accessed.
    double[] data;

    // Determines what part of data to process.
    int start, end;

    Sum(double[] vals, int s, int e) {
        data = vals;
```

```
    start = s;
    end = e;
}

// Find the summation of an array of doubles.
protected Double compute() {
    double sum = 0;

    // If number of elements is below the sequential threshold,
    // then process sequentially.
    if((end - start) < seqThresHold) {
        // Sum the elements.
        for(int i = start; i < end; i++) sum += data[i];
    }
    else {
        // Otherwise, continue to break the data into smaller pieces.

        // Find the midpoint.
        int middle = (start + end) / 2;

        // Invoke new tasks, using the subdivided data.
        Sum subTaskA = new Sum(data, start, middle);
        Sum subTaskB = new Sum(data, middle, end);

        // Start each subtask by forking.
        subTaskA.fork();
        subTaskB.fork();

        // Wait for the subtasks to return, and aggregate the results.
        sum = subTaskA.join() + subTaskB.join();
    }
    // Return the final sum.
    return sum;
}
}

// Demonstrate parallel execution.
class RecurTaskDemo {
    public static void main(String args[]) {
        // Create a task pool.
        ForkJoinPool fjp = new ForkJoinPool();

        double[] nums = new double[5000];

        // Initialize nums with values that alternate between
        // positive and negative.
        for(int i=0; i < nums.length; i++)
            nums[i] = (double) (((i%2) == 0) ? i : -i) ;

        Sum task = new Sum(nums, 0, nums.length);

        // Start the ForkJoinTasks. Notice that, in this case,
        // invoke() returns a result.
        double summation = fjp.invoke(task);
```

```

        System.out.println("Summation " + summation);
    }
}

```

Here's the output from the program:

```
Summation -2500.0
```

There are a couple of interesting items in this program. First, notice that the two subtasks are executed by calling **fork()**, as shown here:

```
subTaskA.fork();
subTaskB.fork();
```

In this case, **fork()** is used because it starts a task but does not wait for it to finish. (Thus, it asynchronously runs the task.) The result of each task is obtained by calling **join()**, as shown here:

```
sum = subTaskA.join() + subTaskB.join();
```

This statement waits until each task ends. It then adds the results of each and assigns the total to **sum**. Thus, the summation of each subtask is added to the running total. Finally, **compute()** ends by returning **sum**, which will be the final total when the first invocation returns.

There are other ways to approach the handling of the asynchronous execution of the subtasks. For example, the following sequence uses **fork()** to start **subTaskA** and uses **invoke()** to start and wait for **subTaskB**:

```
subTaskA.fork();
sum = subTaskB.invoke() + subTaskA.join();
```

Another alternative is to have **subTaskB** call **compute()** directly, as shown here:

```
subTaskA.fork();
sum = subTaskB.compute() + subTaskA.join();
```

Executing a Task Asynchronously

The preceding programs have called **invoke()** on a **ForkJoinPool** to initiate a task. This approach is commonly used when the calling thread must wait until the task has completed (which is often the case) because **invoke()** does not return until the task has terminated. However, you can start a task asynchronously. In this approach, the calling thread continues to execute. Thus, both the calling thread and the task execute simultaneously. To start a task asynchronously, use **execute()**, which is also defined by **ForkJoinPool**. It has the two forms shown here:

```
void execute(ForkJoinTask<?> task)
```

```
void execute(Runnable task)
```

In both forms, *task* specifies the task to run. Notice that the second form lets you specify a **Runnable** rather than a **ForkJoinTask** task. Thus, it forms a bridge between Java's traditional approach to multithreading and the new Fork/Join Framework. It is important to remember that the threads used by a **ForkJoinPool** are daemon. Thus, they will end when the main thread ends. As a result, you may need to keep the main thread alive until the tasks have finished.

Cancelling a Task

A task can be cancelled by calling **cancel()**, which is defined by **ForkJoinTask**. It has this general form:

```
boolean cancel(boolean interruptOK)
```

It returns **true** if the task on which it was called is cancelled. It returns **false** if the task has ended or can't be cancelled. At this time, the *interruptOK* parameter is not used by the default implementation. In general, **cancel()** is intended to be called from code outside the task because a task can easily cancel itself by returning.

You can determine if a task has been cancelled by calling **isCancelled()**, as shown here:

```
final boolean isCancelled()
```

It returns **true** if the invoking task has been cancelled prior to completion and **false** otherwise.

Determining a Task's Completion Status

In addition to **isCancelled()**, which was just described, **ForkJoinTask** includes two other methods that you can use to determine a task's completion status. The first is **isCompletedNormally()**, which is shown here:

```
final boolean isCompletedNormally()
```

It returns **true** if the invoking task completed normally, that is, if it did not throw an exception and it was not cancelled via a call to **cancel()**. It returns **false** otherwise.

The second is **isCompletedAbnormally()**, which is shown here:

```
final boolean isCompletedAbnormally()
```

It returns **true** if the invoking task completed because it was cancelled or because it threw an exception. It returns **false** otherwise.

Restarting a Task

Normally, you cannot rerun a task. In other words, once a task completes, it cannot be restarted. However, you can reinitialize the state of the task (after it has completed) so it can be run again. This is done by calling **reinitialize()**, as shown here:

```
void reinitialize()
```

This method resets the state of the invoking task. However, any modification made to any persistent data that is operated upon by the task will not be undone. For example, if the task modifies an array, then those modifications are not undone by calling **reinitialize()**.

Things to Explore

The preceding discussion presented the fundamentals of the Fork/Join Framework and described several commonly used methods. However, Fork/Join is a rich framework that includes additional capabilities that give you extended control over concurrency. Although it is far beyond the scope of this book to examine all of the issues and nuances surrounding parallel programming and the Fork/Join Framework, a sampling of the other features are mentioned here.

A Sampling of Other `ForkJoinTask` Features

In some cases, you will want to ensure that methods such as `invokeAll()` and `fork()` are called only from within a `ForkJoinTask`. (This may be especially important when using JDK 7, which does not support the common pool.) This is usually a simple matter, but occasionally, you may have code that can be executed from either inside or outside a task. You can determine if your code is executing inside a task by calling `inForkJoinPool()`.

You can convert a `Runnable` or `Callable` object into a `ForkJoinTask` by using the `adapt()` method defined by `ForkJoinTask`. It has three forms, one for converting a `Callable`, one for a `Runnable` that does not return a result, and one for a `Runnable` that does return a result. In the case of a `Callable`, the `call()` method is run. In the case of `Runnable`, the `run()` method is run.

You can obtain an approximate count of the number of tasks that are in the queue of the invoking thread by calling `getQueuedTaskCount()`. You can obtain an approximate count of how many tasks the invoking thread has in its queue that are in excess of the number of other threads in the pool that might “steal” them, by calling `getSurplusQueuedTaskCount()`. Remember, in the Fork/Join Framework, work-stealing is one way in which a high level of efficiency is obtained. Although this process is automatic, in some cases, the information may prove helpful in optimizing through-put.

`ForkJoinTask` defines the following variants of `join()` and `invoke()` that begin with the prefix `quietly`. They are shown here:

<code>final void quietlyJoin()</code>	Joins a task, but does not return a result or throw an exception
<code>final void quietlyInvoke()</code>	Invokes a task, but does not return a result or throw an exception.

In essence, these methods are similar to their non-quiet counterparts except they don’t return values or throw exceptions.

You can attempt to “un-invoke” (in other words, unschedule) a task by calling `tryUnfork()`.

JDK 8 adds several methods, such as `getForkJoinTaskTag()` and `setForkJoinTaskTag()`, that support tags. Tags are short integer values that are linked with a task. They may be useful in specialized applications.

`ForkJoinTask` implements `Serializable`. Thus, it can be serialized. However, serialization is not used during execution.

A Sampling of Other ForkJoinPool Features

One method that is quite useful when tuning fork/join applications is **ForkJoinPool**'s override of **toString()**. It displays a “user-friendly” synopsis of the state of the pool. To see it in action, use this sequence to start and then wait for the task in the **FJExperiment** class of the task experimenter program shown earlier:

```
// Asynchronously start the main ForkJoinTask.  
fjp.execute(task);  
  
// Display the state of the pool while waiting.  
while(!task.isDone()) {  
    System.out.println(fjp);  
}
```

When you run the program, you will see a series of messages on the screen that describe the state of the pool. Here is an example of one. Of course, your output may vary, based on the number of processors, threshold values, task load, and so on.

```
java.util.concurrent.ForkJoinPool@141d683 [Running, parallelism = 2,  
size = 2, active = 0, running = 2, steals = 0, tasks = 0, submissions = 1]
```

You can determine if a pool is currently idle by calling **isQuiescent()**. It returns **true** if the pool has no active threads and **false** otherwise.

You can obtain the number of worker threads currently in the pool by calling **getPoolSize()**. You can obtain an approximate count of the active threads in the pool by calling **getActiveThreadCount()**.

To shut down a pool, call **shutdown()**. Currently active tasks will still be executed, but no new tasks can be started. To stop a pool immediately, call **shutdownNow()**. In this case, an attempt is made to cancel currently active tasks. (It is important to point out, however, that neither of these methods affects the common pool.) You can determine if a pool is shut down by calling **isShutdown()**. It returns **true** if the pool has been shut down and **false** otherwise. To determine if the pool has been shut down and all tasks have been completed, call **isTerminated()**.

Some Fork/Join Tips

Here are a few tips to help you avoid some of the more troublesome pitfalls associated with using the Fork/Join Framework. First, avoid using a sequential threshold that is too low. In general, erring on the high side is better than erring on the low side. If the threshold is too low, more time can be consumed generating and switching tasks than in processing the tasks. Second, usually it is best to use the default level of parallelism. If you specify a smaller number, it may significantly reduce the benefits of using the Fork/Join Framework.

In general, a **ForkJoinTask** should not use synchronized methods or synchronized blocks of code. Also, you will not normally want to have the **compute()** method use other types of synchronization, such as a semaphore. (The new **Phaser** can, however, be used when appropriate because it is compatible with the fork/join mechanism.) Remember, the main idea behind a **ForkJoinTask** is the divide-and-conquer strategy. Such an approach does not normally lend itself to situations in which outside synchronization is needed. Also, avoid situations in which substantial blocking will occur through I/O. Therefore, in general,

a **ForkJoinTask** will not perform I/O. Simply put, to best utilize the Fork/Join Framework, a task should perform a computation that can run without outside blocking or synchronization.

One last point: Except under unusual circumstances, do not make assumptions about the execution environment that your code will run in. This means you should not assume that some specific number of processors will be available, or that the execution characteristics of your program won't be affected by other processes running at the same time.

The Concurrency Utilities Versus Java's Traditional Approach

Given the power and flexibility found in the concurrency utilities, it is natural to ask the following question: Do they replace Java's traditional approach to multithreading and synchronization? The answer is a resounding no! The original support for multithreading and the built-in synchronization features are still the mechanism that should be employed for many, many Java programs, applets, and servlets. For example, **synchronized**, **wait()**, and **notify()** offer elegant solutions to a wide range of problems. However, when extra control is needed, the concurrency utilities are available to handle the chore. Furthermore, the Fork/Join Framework offers a powerful way to integrate parallel programming techniques into your more sophisticated applications.

CHAPTER

29

The Stream API

Of the many new features added by JDK 8, the two that are, arguably, the most important are lambda expressions and the stream API. Lambda expressions were described in Chapter 15. The stream API is described here. As you will see, the stream API is designed with lambda expressions in mind. Moreover, the stream API provides some of the most significant demonstrations of the power that lambdas bring to Java.

Although its design compatibility with lambda expressions is impressive, the key aspect of the stream API is its ability to perform very sophisticated operations that search, filter, map, or otherwise manipulate data. For example, using the stream API, you can construct sequences of actions that resemble, in concept, the type of database queries for which you might use SQL. Furthermore, in many cases, such actions can be performed in parallel, thus providing a high level of efficiency, especially when large data sets are involved. Put simply, the stream API provides a powerful means of handling data in an efficient, yet easy to use way.

Before continuing, an important point needs to be made: The stream API uses some of Java's most advanced features. To fully understand and utilize it requires a solid understanding of generics and lambda expressions. The basic concepts of parallel execution and a working knowledge of the Collections Framework are also needed. (See Chapters 14, 15, 18, and 28.)

Stream Basics

Let's begin by defining the term *stream* as it applies to the stream API: a stream is a conduit for data. Thus, a stream represents a sequence of objects. A stream operates on a data source, such as an array or a collection. A stream, itself, never provides storage for the data. It simply moves data, possibly filtering, sorting, or otherwise operating on that data in the process. As a general rule, however, a stream operation by itself does not modify the data source. For example, sorting a stream does not change the order of the source. Rather, sorting a stream results in the creation of a new stream that produces the sorted result.

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 <i>handler</i>)	Returns a new stream with the close handler specified by <i>handler</i> . 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 <i>collectorFunc</i> 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 <i>pred</i> . (Intermediate operation.)
void forEach(Consumer<? super T> action)	For each element in the invoking stream, the code specified by <i>action</i> is executed. (Terminal operation.)
<R> Stream<R> map(Function<? super T, ? extends R> mapFunc)	Applies <i>mapFunc</i> to the elements from the invoking stream, yielding a new stream that contains those elements. (Intermediate operation.)
DoubleStream mapToDouble(ToDoubleFunction<? super T> mapFunc)	Applies <i>mapFunc</i> to the elements from the invoking stream, yielding a new DoubleStream that contains those elements. (Intermediate operation.)
IntStream mapToInt(ToIntFunction<? super T> mapFunc)	Applies <i>mapFunc</i> to the elements from the invoking stream, yielding a new IntStream that contains those elements. (Intermediate operation.)
LongStream mapToLong(ToLongFunction<? super T> mapFunc)	Applies <i>mapFunc</i> 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 <i>comp</i> , 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 <i>comp</i> , finds and returns the minimum element in the invoking stream. (Terminal operation.)
T reduce(T identityVal, 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 **Integers**. 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 = myStreamspliterator();

// 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 = myStreamspliterator();  
  
// 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 **isX** 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 <i>val</i>)	Returns true if <i>val</i> has the abstract flag set and false otherwise.
static boolean isFinal(int <i>val</i>)	Returns true if <i>val</i> has the final flag set and false otherwise.
static boolean isInterface(int <i>val</i>)	Returns true if <i>val</i> has the interface flag set and false otherwise.
static boolean isNative(int <i>val</i>)	Returns true if <i>val</i> has the native flag set and false otherwise.
static boolean isPrivate(int <i>val</i>)	Returns true if <i>val</i> has the private flag set and false otherwise.
static boolean isProtected(int <i>val</i>)	Returns true if <i>val</i> has the protected flag set and false otherwise.
static boolean isPublic(int <i>val</i>)	Returns true if <i>val</i> has the public flag set and false otherwise.
static boolean isStatic(int <i>val</i>)	Returns true if <i>val</i> has the static flag set and false otherwise.
static boolean isStrict(int <i>val</i>)	Returns true if <i>val</i> has the strict flag set and false otherwise.
static boolean isSynchronized(int <i>val</i>)	Returns true if <i>val</i> has the synchronized flag set and false otherwise.
static boolean isTransient(int <i>val</i>)	Returns true if <i>val</i> has the transient flag set and false otherwise.
static boolean isVolatile(int <i>val</i>)	Returns true if <i>val</i> 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 **getInstance()** method returns an instance of **DateFormat** that can format date information. It is available in these forms:

```
static final DateFormat getInstance( )
static final DateFormat getInstance(int style)
static final DateFormat getInstance(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.SimpleDateFormat**, described earlier in this chapter.)

Here is an example that uses **DateTimeFormatter** to display the current date and time:

```
// Demonstrate DateTimeFormatter.
import java.time.*;
import java.time.format.*;

class DateTimeDemo2 {
    public static void main(String args[]) {

        LocalDate curDate = LocalDate.now();
        System.out.println(curDate.format(
            DateTimeFormatter.ofLocalizedDate(FormatStyle.FULL)),

        LocalTime curTime = LocalTime.now();
        System.out.println(curTime.format(
            DateTimeFormatter.ofLocalizedTime(FormatStyle.SHORT))) ;
    }
}
```

Sample output is shown here:

```
Wednesday, January 1, 2014
2:16 PM
```

In some situations, you may want a format different from the ones you can specify by use of **FormatStyle**. One way to accomplish this is to use a predefined formatter, such as **ISO_DATE** or **ISO_TIME**, provided by **DateTimeFormatter**. Another way is to create a custom format by specifying a pattern. To do this, you can use the **ofPattern()** factory method of **DateTimeFormatter**. One version is shown here:

```
static DateTimeFormatter ofPattern(String fmtPattern)
```

Here, *fmtPattern* specifies a string that contains the date and time pattern that you want. It returns a **DateTimeFormatter** that will format according to that pattern. The default locale is used.

In general, a pattern consists of format specifiers, called *pattern letters*. A pattern letter will be replaced by the date or time component that it specifies. The full list of pattern letters is shown in the API documentation for **ofPattern()**. Here is a sampling. Note that the pattern letters are case-sensitive.

a	AM/PM indicator
d	Day in month
E	Day in week
h	Hour, 12-hour clock
H	Hour, 24-hour clock
M	Month
m	Minutes
s	Seconds
y	Year

In general, the precise output that you see will be determined by how many times a pattern letter is repeated. (Thus, **DateTimeFormatter** works a bit like **java.text.SimpleDateFormat**, described earlier in this chapter.) For example, assuming that the month is April, the patterns:

M MM MMM MMMM

produce the following formatted output:

4 04 Apr April

Frankly, experimentation is the best way to understand what each pattern letter does and how various repetitions affect the output.

When you want to output a pattern letter as text, enclose the text between single quotation marks. In general, it is a good idea to enclose all non-pattern characters within single quotation marks to avoid problems if the set of pattern letters changes in subsequent versions of Java.

The following program demonstrates the use of a date and time pattern:

```
// Create a custom date and time format.
import java.time.*;
import java.time.format.*;

class DateTimeDemo3 {
    public static void main(String args[]) {

        LocalDateTime curDateTime = LocalDateTime.now();
        System.out.println(curDateTime.format(
            DateTimeFormatter.ofPattern("MMMM d', ' yyyy h': 'mm a")));
    }
}
```

Sample output is shown here:

January 1, 2014 2:22 PM

One other point about creating custom date and time output: **LocalDate**, **LocalTime**, and **LocalDateTime** define methods that let you obtain various date and time components. For example, **getHour()** returns the hour as an **int**; **getMonth()** returns the month in the form of a **Month** enumeration value; and **getYear()** returns the year as an **int**. Using these, and other methods, you can manually construct output. You can also use these values for other purposes, such as when creating specialized timers.

Parsing Date and Time Strings

LocalDate, **LocalTime**, and **LocalDateTime** provide the ability to parse date and/or time strings. To do this, call **parse()** on an instance of one of those classes. It has two forms. The first uses the default formatter that parses the date and/or time formatted in the standard ISO fashion, such as 03:31 for time and 2014-08-02 for date. The form of this version of

parse() for **LocalDateTime** is shown here. (Its form for the other classes is similar except for the type of object returned.)

```
static LocalDateTime parse(CharSequence dateTimeStr)
```

Here, *dateTimeStr* is a string that contains the date and time in the proper format. If the format is invalid, an error will result.

If you want to parse a date and/or time string that is in a format other than ISO format, you can use a second form of **parse()** that lets you specify your own formatter. The version specified by **LocalDateTime** is shown next. (The other classes provide a similar form except for the return type.)

```
static LocalDateTime parse(CharSequence dateTimeStr,
                           DateTimeFormatter dateTimeFmtr)
```

Here, *dateTimeFmtr* specifies the formatter that you want to use.

Here is a simple example that parses a date and time string by use of a custom formatter:

```
// Parse a date and time.
import java.time.*;
import java.time.format.*;

class DateTimeDemo4 {
    public static void main(String args[]) {

        // Obtain a LocalDateTime object by parsing a date and time string.
        LocalDateTime curDateTime =
            LocalDateTime.parse("June 21, 2014 12:01 AM",
                               DateTimeFormatter.ofPattern("MMMM d', ' yyyy hh':'mm a"));

        // Now, display the parsed date and time.
        System.out.println(curDateTime.format(
            DateTimeFormatter.ofPattern("MMMM d', ' yyyy hh':'mm a")));
    }
}
```

Sample output is shown here:

```
June 21, 2014 12:01 AM
```

Other Things to Explore in `java.time`

Although you will want to explore all of the date and time packages, a good place to start is with **java.time**. It contains a great deal of functionality that you may find useful. Begin by examining the methods defined by **LocalDate**, **LocalTime**, and **LocalDateTime**. Each has methods that let you add or subtract dates and/or times, adjust dates and/or times by a given component, compare dates and/or times, and create instances based on date and/or time components, among others. Other classes in **java.time** that you may find of particular interest include **Instant**, **Duration**, and **Period**. **Instant** encapsulates an instant in time. **Duration** encapsulates a length of time. **Period** encapsulates a length of date.

PART

III

Introducing GUI Programming with Swing

CHAPTER 31

Introducing Swing

CHAPTER 32

Exploring Swing

CHAPTER 33

Introducing Swing Menus

This page has been intentionally left blank

CHAPTER

31

Introducing Swing

In Part II, you saw how to build very simple user interfaces with the AWT classes. Although the AWT is still a crucial part of Java, its component set is no longer widely used to create graphical user interfaces. Today, most programmers use Swing or JavaFX for this purpose. JavaFX is discussed in Part IV. Here, Swing is introduced. Swing is a framework that provides more powerful and flexible GUI components than does the AWT. As a result, it is the GUI that has been widely used by Java programmers for more than a decade.

Coverage of Swing is divided between three chapters. This chapter introduces Swing. It begins by describing Swing's core concepts. It then shows the general form of a Swing program, including both applications and applets. It concludes by explaining how painting is accomplished in Swing. The next chapter presents several commonly used Swing components. The third chapter introduces Swing-based menus. It is important to understand that the number of classes and interfaces in the Swing packages is quite large, and they can't all be covered in this book. (In fact, full coverage of Swing requires an entire book of its own.) However, these three chapters will give you a basic understanding of this important topic.

NOTE For a comprehensive introduction to Swing, see my book *Swing: A Beginner's Guide* published by McGraw-Hill Professional (2007).

The Origins of Swing

Swing did not exist in the early days of Java. Rather, it was a response to deficiencies present in Java's original GUI subsystem: the Abstract Window Toolkit. The AWT defines a basic set of controls, windows, and dialog boxes that support a usable, but limited graphical interface. One reason for the limited nature of the AWT is that it translates its various visual components into their corresponding, platform-specific equivalents, or *peers*. This means that the look and feel of a component is defined by the platform, not by Java. Because the AWT components use native code resources, they are referred to as *heavyweight*.

The use of native peers led to several problems. First, because of variations between operating systems, a component might look, or even act, differently on different platforms.

This potential variability threatened the overarching philosophy of Java: write once, run anywhere. Second, the look and feel of each component was fixed (because it is defined by the platform) and could not be (easily) changed. Third, the use of heavyweight components caused some frustrating restrictions. For example, a heavyweight component was always opaque.

Not long after Java's original release, it became apparent that the limitations and restrictions present in the AWT were sufficiently serious that a better approach was needed. The solution was Swing. Introduced in 1997, Swing was included as part of the Java Foundation Classes (JFC). Swing was initially available for use with Java 1.1 as a separate library. However, beginning with Java 1.2, Swing (and the rest of the JFC) was fully integrated into Java.

Swing Is Built on the AWT

Before moving on, it is necessary to make one important point: although Swing eliminates a number of the limitations inherent in the AWT, Swing *does not* replace it. Instead, Swing is built on the foundation of the AWT. This is why the AWT is still a crucial part of Java. Swing also uses the same event handling mechanism as the AWT. Therefore, a basic understanding of the AWT and of event handling is required to use Swing. (The AWT is covered in Chapters 25 and 26. Event handling is described in Chapter 24.)

Two Key Swing Features

As just explained, Swing was created to address the limitations present in the AWT. It does this through two key features: lightweight components and a pluggable look and feel. Together they provide an elegant, yet easy-to-use solution to the problems of the AWT. More than anything else, it is these two features that define the essence of Swing. Each is examined here.

Swing Components Are Lightweight

With very few exceptions, Swing components are *lightweight*. This means that they are written entirely in Java and do not map directly to platform-specific peers. Thus, lightweight components are more efficient and more flexible. Furthermore, because lightweight components do not translate into native peers, the look and feel of each component is determined by Swing, not by the underlying operating system. As a result, each component will work in a consistent manner across all platforms.

Swing Supports a Pluggable Look and Feel

Swing supports a *pluggable look and feel* (PLAF). Because each Swing component is rendered by Java code rather than by native peers, the look and feel of a component is under the control of Swing. This fact means that it is possible to separate the look and feel of a component from the logic of the component, and this is what Swing does. Separating out the look and feel provides a significant advantage: it becomes possible to change the way that a component is rendered without affecting any of its other aspects. In other words, it is possible to “plug in” a new look and feel for any given component without creating any side effects in the code that uses that component. Moreover, it becomes possible to define entire

sets of look-and-feels that represent different GUI styles. To use a specific style, its look and feel is simply “plugged in.” Once this is done, all components are automatically rendered using that style.

Pluggable look-and-feels offer several important advantages. It is possible to define a look and feel that is consistent across all platforms. Conversely, it is possible to create a look and feel that acts like a specific platform. For example, if you know that an application will be running only in a Windows environment, it is possible to specify the Windows look and feel. It is also possible to design a custom look and feel. Finally, the look and feel can be changed dynamically at run time.

Java 8 provides look-and-feels, such as metal and Nimbus, that are available to all Swing users. The metal look and feel is also called the *Java look and feel*. It is platform-independent and available in all Java execution environments. It is also the default look and feel. Windows environments also have access to the Windows look and feel. This book uses the default Java look and feel (metal) because it is platform independent.

The MVC Connection

In general, a visual component is a composite of three distinct aspects:

- The way that the component looks when rendered on the screen
- The way that the component reacts to the user
- The state information associated with the component

No matter what architecture is used to implement a component, it must implicitly contain these three parts. Over the years, one component architecture has proven itself to be exceptionally effective: *Model-View-Controller*, or MVC for short.

The MVC architecture is successful because each piece of the design corresponds to an aspect of a component. In MVC terminology, the *model* corresponds to the state information associated with the component. For example, in the case of a check box, the model contains a field that indicates if the box is checked or unchecked. The *view* determines how the component is displayed on the screen, including any aspects of the view that are affected by the current state of the model. The *controller* determines how the component reacts to the user. For example, when the user clicks a check box, the controller reacts by changing the model to reflect the user’s choice (checked or unchecked). This then results in the view being updated. By separating a component into a model, a view, and a controller, the specific implementation of each can be changed without affecting the other two. For instance, different view implementations can render the same component in different ways without affecting the model or the controller.

Although the MVC architecture and the principles behind it are conceptually sound, the high level of separation between the view and the controller is not beneficial for Swing components. Instead, Swing uses a modified version of MVC that combines the view and the controller into a single logical entity called the *UI delegate*. For this reason, Swing’s approach is called either the *Model-Delegate* architecture or the *Separable Model* architecture. Therefore, although Swing’s component architecture is based on MVC, it does not use a classical implementation of it.

Swing's pluggable look and feel is made possible by its Model-Delegate architecture. Because the view (look) and controller (feel) are separate from the model, the look and feel can be changed without affecting how the component is used within a program. Conversely, it is possible to customize the model without affecting the way that the component appears on the screen or responds to user input.

To support the Model-Delegate architecture, most Swing components contain two objects. The first represents the model. The second represents the UI delegate. Models are defined by interfaces. For example, the model for a button is defined by the **ButtonModel** interface. UI delegates are classes that inherit **ComponentUI**. For example, the UI delegate for a button is **ButtonUI**. Normally, your programs will not interact directly with the UI delegate.

Components and Containers

A Swing GUI consists of two key items: *components* and *containers*. However, this distinction is mostly conceptual because all containers are also components. The difference between the two is found in their intended purpose: As the term is commonly used, a *component* is an independent visual control, such as a push button or slider. A container holds a group of components. Thus, a container is a special type of component that is designed to hold other components. Furthermore, in order for a component to be displayed, it must be held within a container. Thus, all Swing GUIs will have at least one container. Because containers are components, a container can also hold other containers. This enables Swing to define what is called a *containment hierarchy*, at the top of which must be a *top-level container*.

Let's look a bit more closely at components and containers.

Components

In general, Swing components are derived from the **JComponent** class. (The only exceptions to this are the four top-level containers, described in the next section.) **JComponent** provides the functionality that is common to all components. For example, **JComponent** supports the pluggable look and feel. **JComponent** inherits the AWT classes **Container** and **Component**. Thus, a Swing component is built on and compatible with an AWT component.

All of Swing's components are represented by classes defined within the package **javax.swing**. The following table shows the class names for Swing components (including those used as containers).

JApplet	JButton	JCheckBox	JCheckBoxMenuItem
JColorChooser	JComboBox	JComponent	JDesktopPane
JDialog	JEditorPane	JFileChooser	JFormattedTextField
JFrame	JInternalFrame	JLabel	JLayer
JLayeredPane	JList	JMenu	JMenuBar
JMenuItem	JOptionPane	JPanel	JPasswordField
JPopupMenu	JProgressBar	JRadioButton	JRadioButtonMenuItem
JRootPane	JScrollBar	JScrollPane	JSeparator
JSlider	JSpinner	JSplitPane	JTabbedPane

JTable	JTextArea	JTextField	JTextPane
JToggleButton	JToolBar	JToolTip	JTree
JViewport	JWindow		

Notice that all component classes begin with the letter **J**. For example, the class for a label is **JLabel**; the class for a push button is **JButton**; and the class for a scroll bar is **JScrollBar**.

Containers

Swing defines two types of containers. The first are top-level containers: **JFrame**, **JApplet**, **JWindow**, and **JDialog**. These containers do not inherit **JComponent**. They do, however, inherit the AWT classes **Component** and **Container**. Unlike Swing's other components, which are lightweight, the top-level containers are heavyweight. This makes the top-level containers a special case in the Swing component library.

As the name implies, a top-level container must be at the top of a containment hierarchy. A top-level container is not contained within any other container. Furthermore, every containment hierarchy must begin with a top-level container. The one most commonly used for applications is **JFrame**. The one used for applets is **JApplet**.

The second type of containers supported by Swing are lightweight containers. Lightweight containers *do* inherit **JComponent**. An example of a lightweight container is **JPanel**, which is a general-purpose container. Lightweight containers are often used to organize and manage groups of related components because a lightweight container can be contained within another container. Thus, you can use lightweight containers such as **JPanel** to create subgroups of related controls that are contained within an outer container.

The Top-Level Container Panes

Each top-level container defines a set of *panes*. At the top of the hierarchy is an instance of **JRootPane**. **JRootPane** is a lightweight container whose purpose is to manage the other panes. It also helps manage the optional menu bar. The panes that comprise the root pane are called the *glass pane*, the *content pane*, and the *layered pane*.

The glass pane is the top-level pane. It sits above and completely covers all other panes. By default, it is a transparent instance of **JPanel**. The glass pane enables you to manage mouse events that affect the entire container (rather than an individual control) or to paint over any other component, for example. In most cases, you won't need to use the glass pane directly, but it is there if you need it.

The layered pane is an instance of **JLayeredPane**. The layered pane allows components to be given a depth value. This value determines which component overlays another. (Thus, the layered pane lets you specify a Z-order for a component, although this is not something that you will usually need to do.) The layered pane holds the content pane and the (optional) menu bar.

Although the glass pane and the layered panes are integral to the operation of a top-level container and serve important purposes, much of what they provide occurs behind the scene. The pane with which your application will interact the most is the content pane, because this is the pane to which you will add visual components. In other words, when you add a component, such as a button, to a top-level container, you will add it to the content pane. By default, the content pane is an opaque instance of **JPanel**.

The Swing Packages

Swing is a very large subsystem and makes use of many packages. At the time of this writing, these are the packages defined by Swing.

javax.swing	javax.swing.plaf.basic	javax.swing.text
javax.swing.border	javax.swing.plaf.metal	javax.swing.text.html
javax.swing.colorchooser	javax.swing.plaf.multi	javax.swing.text.html.parser
javax.swing.event	javax.swing.plaf.nimbus	javax.swing.text.rtf
javax.swing.filechooser	javax.swing.plaf.synth	javax.swing.tree
javax.swing.plaf	javax.swing.table	javax.swing.undo

The main package is **javax.swing**. This package must be imported into any program that uses Swing. It contains the classes that implement the basic Swing components, such as push buttons, labels, and check boxes.

A Simple Swing Application

Swing programs differ from both the console-based programs and the AWT-based programs shown earlier in this book. For example, they use a different set of components and a different container hierarchy than does the AWT. Swing programs also have special requirements that relate to threading. The best way to understand the structure of a Swing program is to work through an example. There are two types of Java programs in which Swing is typically used. The first is a desktop application. The second is the applet. This section shows how to create a Swing application. The creation of a Swing applet is described later in this chapter.

Although quite short, the following program shows one way to write a Swing application. In the process, it demonstrates several key features of Swing. It uses two Swing components: **JFrame** and **JLabel**. **JFrame** is the top-level container that is commonly used for Swing applications. **JLabel** is the Swing component that creates a label, which is a component that displays information. The label is Swing's simplest component because it is passive. That is, a label does not respond to user input. It just displays output. The program uses a **JFrame** container to hold an instance of a **JLabel**. The label displays a short text message.

```
// A simple Swing application.

import javax.swing.*;

class SwingDemo {

    SwingDemo() {

        // Create a new JFrame container.
        JFrame jfrm = new JFrame("A Simple Swing Application");
    }
}
```

```
// Give the frame an initial size.  
jfrm.setSize(275, 100);  
  
// Terminate the program when the user closes the application.  
jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
  
// Create a text-based label.  
JLabel jlab = new JLabel(" Swing means powerful GUIs.");  
  
// Add the label to the content pane.  
jfrm.add(jlab);  
  
// Display the frame.  
jfrm.setVisible(true);  
}  
  
public static void main(String args[]) {  
    // Create the frame on the event dispatching thread.  
    SwingUtilities.invokeLater(new Runnable() {  
        public void run() {  
            new SwingDemo();  
        }  
    });  
}
```

Swing programs are compiled and run in the same way as other Java applications. Thus, to compile this program, you can use this command line:

```
javac SwingDemo.java
```

To run the program, use this command line:

```
java SwingDemo
```

When the program is run, it will produce a window similar to that shown in Figure 31-1.

Because the **SwingDemo** program illustrates several core Swing concepts, we will examine it carefully, line by line. The program begins by importing **javax.swing**. As mentioned, this package contains the components and models defined by Swing. For example, **javax.swing** defines classes that implement labels, buttons, text controls, and menus. It will be included in all programs that use Swing.



Figure 31-1 The window produced by the **SwingDemo** program

Next, the program declares the **SwingDemo** class and a constructor for that class. The constructor is where most of the action of the program occurs. It begins by creating a **JFrame**, using this line of code:

```
jfrm = new JFrame("A Simple Swing Application");
```

This creates a container called **jfrm** that defines a rectangular window complete with a title bar; close, minimize, maximize, and restore buttons; and a system menu. Thus, it creates a standard, top-level window. The title of the window is passed to the constructor.

Next, the window is sized using this statement:

```
jfrm.setSize(275, 100);
```

The **setSize()** method (which is inherited by **JFrame** from the AWT class **Component**) sets the dimensions of the window, which are specified in pixels. Its general form is shown here:

```
void setSize(int width, int height)
```

In this example, the width of the window is set to 275 and the height is set to 100.

By default, when a top-level window is closed (such as when the user clicks the close box), the window is removed from the screen, but the application is not terminated.

While this default behavior is useful in some situations, it is not what is needed for most applications. Instead, you will usually want the entire application to terminate when its top-level window is closed. There are a couple of ways to achieve this. The easiest way is to call **setDefaultCloseOperation()**, as the program does:

```
jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

After this call executes, closing the window causes the entire application to terminate. The general form of **setDefaultCloseOperation()** is shown here:

```
void setDefaultCloseOperation(int what)
```

The value passed in *what* determines what happens when the window is closed. There are several other options in addition to **JFrame.EXIT_ON_CLOSE**. They are shown here:

DISPOSE_ON_CLOSE

HIDE_ON_CLOSE

DO NOTHING ON CLOSE

Their names reflect their actions. These constants are declared in **WindowConstants**, which is an interface declared in **javax.swing** that is implemented by **JFrame**.

The next line of code creates a Swing **JLabel** component:

```
JLabel jlab = new JLabel(" Swing means powerful GUIs.");
```

JLabel is the simplest and easiest-to-use component because it does not accept user input. It simply displays information, which can consist of text, an icon, or a combination of the two. The label created by the program contains only text, which is passed to its constructor.

The next line of code adds the label to the content pane of the frame:

```
jfrm.add(jlab);
```

As explained earlier, all top-level containers have a content pane in which components are stored. Thus, to add a component to a frame, you must add it to the frame's content pane. This is accomplished by calling `add()` on the **JFrame** reference (`jfrm` in this case). The general form of `add()` is shown here:

Component `add(Component comp)`

The `add()` method is inherited by **JFrame** from the AWT class **Container**.

By default, the content pane associated with a **JFrame** uses border layout. The version of `add()` just shown adds the label to the center location. Other versions of `add()` enable you to specify one of the border regions. When a component is added to the center, its size is adjusted automatically to fit the size of the center.

Before continuing, an important historical point needs to be made. Prior to JDK 5, when adding a component to the content pane, you could not invoke the `add()` method directly on a **JFrame** instance. Instead, you needed to call `add()` on the content pane of the **JFrame** object. The content pane can be obtained by calling `getContentPane()` on a **JFrame** instance. The `getContentPane()` method is shown here:

Container `getContentPane()`

It returns a **Container** reference to the content pane. The `add()` method was then called on that reference to add a component to a content pane. Thus, in the past, you had to use the following statement to add `jlab` to `jfrm`:

```
jfrm.getContentPane().add(jlab); // old-style
```

Here, `getContentPane()` first obtains a reference to content pane, and then `add()` adds the component to the container linked to this pane. This same procedure was also required to invoke `remove()` to remove a component and `setLayout()` to set the layout manager for the content pane. You will see explicit calls to `getContentPane()` frequently throughout pre-5.0 code. Today, the use of `getContentPane()` is no longer necessary. You can simply call `add()`, `remove()`, and `setLayout()` directly on **JFrame** because these methods have been changed so that they operate on the content pane automatically.

The last statement in the **SwingDemo** constructor causes the window to become visible:

```
jfrm.setVisible(true);
```

The `setVisible()` method is inherited from the AWT **Component** class. If its argument is `true`, the window will be displayed. Otherwise, it will be hidden. By default, a **JFrame** is invisible, so `setVisible(true)` must be called to show it.

Inside `main()`, a **SwingDemo** object is created, which causes the window and the label to be displayed. Notice that the **SwingDemo** constructor is invoked using these lines of code:

```
SwingUtilities.invokeLater(new Runnable() {
    public void run() {
        new SwingDemo();
    }
});
```

This sequence causes a **SwingDemo** object to be created on the *event dispatching thread* rather than on the main thread of the application. Here's why. In general, Swing programs are event-driven. For example, when a user interacts with a component, an event is

generated. An event is passed to the application by calling an event handler defined by the application. However, the handler is executed on the event dispatching thread provided by Swing and not on the main thread of the application. Thus, although event handlers are defined by your program, they are called on a thread that was not created by your program.

To avoid problems (including the potential for deadlock), all Swing GUI components must be created and updated from the event dispatching thread, not the main thread of the application. However, `main()` is executed on the main thread. Thus, `main()` cannot directly instantiate a `SwingDemo` object. Instead, it must create a `Runnable` object that executes on the event dispatching thread and have this object create the GUI.

To enable the GUI code to be created on the event dispatching thread, you must use one of two methods that are defined by the `SwingUtilities` class. These methods are `invokeLater()` and `invokeAndWait()`. They are shown here:

```
static void invokeLater(Runnable obj)
static void invokeAndWait(Runnable obj)
    throws InterruptedException, InvocationTargetException
```

Here, `obj` is a `Runnable` object that will have its `run()` method called by the event dispatching thread. The difference between the two methods is that `invokeLater()` returns immediately, but `invokeAndWait()` waits until `obj.run()` returns. You can use one of these methods to call a method that constructs the GUI for your Swing application, or whenever you need to modify the state of the GUI from code not executed by the event dispatching thread. You will normally want to use `invokeLater()`, as the preceding program does. However, when constructing the initial GUI for an applet, you will need to use `invokeAndWait()`.

Event Handling

The preceding example showed the basic form of a Swing program, but it left out one important part: event handling. Because `JLabel` does not take input from the user, it does not generate events, so no event handling was needed. However, the other Swing components *do* respond to user input and the events generated by those interactions need to be handled. Events can also be generated in ways not directly related to user input. For example, an event is generated when a timer goes off. Whatever the case, event handling is a large part of any Swing-based application.

The event handling mechanism used by Swing is the same as that used by the AWT. This approach is called the *delegation event model*, and it is described in Chapter 24. In many cases, Swing uses the same events as does the AWT, and these events are packaged in `java.awt.event`. Events specific to Swing are stored in `javax.swing.event`.

Although events are handled in Swing in the same way as they are with the AWT, it is still useful to work through a simple example. The following program handles the event generated by a Swing push button. Sample output is shown in Figure 31-2.



Figure 31-2 Output from the `EventDemo` program

```
// Handle an event in a Swing program.

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

class EventDemo {

    JLabel jlab;

    EventDemo() {

        // Create a new JFrame container.
        JFrame jfrm = new JFrame("An Event Example");

        // Specify FlowLayout for the layout manager.
        jfrm.setLayout(new FlowLayout());

        // Give the frame an initial size.
        jfrm.setSize(220, 90);

        // Terminate the program when the user closes the application.
        jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        // Make two buttons.
        JButton jbtnAlpha = new JButton("Alpha");
        JButton jbtnBeta = new JButton("Beta");

        // Add action listener for Alpha.
        jbtnAlpha.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent ae) {
                jlab.setText("Alpha was pressed.");
            }
        });

        // Add action listener for Beta.
        jbtnBeta.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent ae) {
                jlab.setText("Beta was pressed.");
            }
        });

        // Add the buttons to the content pane.
        jfrm.add(jbtnAlpha);
        jfrm.add(jbtnBeta);

        // Create a text-based label.
        jlab = new JLabel("Press a button.");

        // Add the label to the content pane.
        jfrm.add(jlab);

        // Display the frame.
        jfrm.setVisible(true);
    }
}
```

```

public static void main(String args[]) {
    // Create the frame on the event dispatching thread.
    SwingUtilities.invokeLater(new Runnable() {
        public void run() {
            new EventDemo();
        }
    });
}
}

```

First, notice that the program now imports both the **java.awt** and **java.awt.event** packages. The **java.awt** package is needed because it contains the **FlowLayout** class, which supports the standard flow layout manager used to lay out components in a frame. (See Chapter 26 for coverage of layout managers.) The **java.awt.event** package is needed because it defines the **ActionListener** interface and the **ActionEvent** class.

The **EventDemo** constructor begins by creating a **JFrame** called **jfrm**. It then sets the layout manager for the content pane of **jfrm** to **FlowLayout**. Recall that, by default, the content pane uses **BorderLayout** as its layout manager. However, for this example, **FlowLayout** is more convenient. Notice that **FlowLayout** is assigned using this statement:

```
jfrm.setLayout(new FlowLayout());
```

As explained, in the past you had to explicitly call **getContentPane()** to set the layout manager for the content pane. This requirement was removed as of JDK 5.

After setting the size and default close operation, **EventDemo()** creates two push buttons, as shown here:

```

 JButton jbtnAlpha = new JButton("Alpha");
 JButton jbtnBeta = new JButton("Beta");

```

The first button will contain the text "Alpha" and the second will contain the text "Beta". Swing push buttons are instances of **JButton**. **JButton** supplies several constructors. The one used here is

```
JButton(String msg)
```

The *msg* parameter specifies the string that will be displayed inside the button.

When a push button is pressed, it generates an **ActionEvent**. Thus, **JButton** provides the **addActionListener()** method, which is used to add an action listener. (**JButton** also provides **removeActionListener()** to remove a listener, but this method is not used by the program.) As explained in Chapter 24, the **ActionListener** interface defines only one method: **actionPerformed()**. It is shown again here for your convenience:

```
void actionPerformed(ActionEvent ae)
```

This method is called when a button is pressed. In other words, it is the event handler that is called when a button press event has occurred.

Next, event listeners for the button's action events are added by the code shown here:

```

// Add action listener for Alpha.
jbtnAlpha.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent ae) {

```

```

        jlab.setText("Alpha was pressed.");
    }
});

// Add action listener for Beta.
jbtnBeta.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent ae) {
        jlab.setText("Beta was pressed.");
    }
});

```

Here, anonymous inner classes are used to provide the event handlers for the two buttons. Each time a button is pressed, the string displayed in **jlab** is changed to reflect which button was pressed.

Beginning with JDK 8, lambda expressions can also be used to implement event handlers. For example, the event handler for the Alpha button could be written like this:

```
jbtnAlpha.addActionListener( (ae) -> jlab.setText("Alpha was pressed."));
```

As you can see, this code is shorter. For the benefit of readers using versions of Java prior to JDK 8, subsequent examples will not use lambda expressions, but you should consider using them for new code that you create.

Next, the buttons are added to the content pane of **jfrm**:

```
jfrm.add(jbtnAlpha);
jfrm.add(jbtnBeta);
```

Finally, **jlab** is added to the content pane and window is made visible. When you run the program, each time you press a button, a message is displayed in the label that indicates which button was pressed.

One last point: Remember that all event handlers, such as **actionPerformed()**, are called on the event dispatching thread. Therefore, an event handler must return quickly in order to avoid slowing down the application. If your application needs to do something time consuming as the result of an event, it must use a separate thread.

Create a Swing Applet

The second type of program that commonly uses Swing is the applet. Swing-based applets are similar to AWT-based applets, but with an important difference: A Swing applet extends **JApplet** rather than **Applet**. **JApplet** is derived from **Applet**. Thus, **JApplet** includes all of the functionality found in **Applet** and adds support for Swing. **JApplet** is a top-level Swing container, which means that it is *not* derived from **JComponent**. Because **JApplet** is a top-level container, it includes the various panes described earlier. This means that all components are added to **JApplet**'s content pane in the same way that components are added to **JFrame**'s content pane.

Swing applets use the same four life-cycle methods as described in Chapter 23: **init()**, **start()**, **stop()**, and **destroy()**. Of course, you need override only those methods that are needed by your applet. Painting is accomplished differently in Swing than it is in the AWT, and a Swing applet will not normally override the **paint()** method. (Painting in Swing is described later in this chapter.)



Figure 31-3 Output from the example Swing applet

One other point: All interaction with components in a Swing applet must take place on the event dispatching thread, as described in the previous section. This threading issue applies to all Swing programs.

Here is an example of a Swing applet. It provides the same functionality as the previous application, but does so in applet form. Figure 31-3 shows the program when executed by **appletviewer**.

```
// A simple Swing-based applet

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

/*
This HTML can be used to launch the applet:

<applet code="MySwingApplet" width=220 height=90>
</applet>
*/

public class MySwingApplet extends JApplet {
    JButton jbtnAlpha;
    JButton jbtnBeta;

    JLabel jlab;

    // Initialize the applet.
    public void init() {
        try {
            SwingUtilities.invokeAndWait(new Runnable () {
                public void run() {
                    makeGUI(); // initialize the GUI
                }
            });
        } catch(Exception exc) {
            System.out.println("Can't create because of "+ exc);
        }
    }
}
```

```
// This applet does not need to override start(), stop(),
// or destroy().

// Set up and initialize the GUI.
private void makeGUI() {

    // Set the applet to use flow layout.
    setLayout(new FlowLayout());

    // Make two buttons.
    jbtnAlpha = new JButton("Alpha");
    jbtnBeta = new JButton("Beta");

    // Add action listener for Alpha.
    jbtnAlpha.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent le) {
            jlab.setText("Alpha was pressed.");
        }
    });

    // Add action listener for Beta.
    jbtnBeta.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent le) {
            jlab.setText("Beta was pressed.");
        }
    });

    // Add the buttons to the content pane.
    add(jbtnAlpha);
    add(jbtnBeta);

    // Create a text-based label.
    jlab = new JLabel("Press a button.");

    // Add the label to the content pane.
    add(jlab);
}
}
```

There are two important things to notice about this applet. First, **MySwingApplet** extends **JApplet**. As explained, all Swing-based applets extend **JApplet** rather than **Applet**. Second, the **init()** method initializes the Swing components on the event dispatching thread by setting up a call to **makeGUI()**. Notice that this is accomplished through the use of **invokeAndWait()** rather than **invokeLater()**. Applets must use **invokeAndWait()** because the **init()** method must not return until the entire initialization process has been completed. In essence, the **start()** method cannot be called until after initialization, which means that the GUI must be fully constructed.

Inside **makeGUI()**, the two buttons and label are created, and the action listeners are added to the buttons. Finally, the components are added to the content pane. Although this example is quite simple, this same general approach must be used when building any Swing GUI that will be used by an applet.

Painting in Swing

Although the Swing component set is quite powerful, you are not limited to using it because Swing also lets you write directly into the display area of a frame, panel, or one of Swing's other components, such as **JLabel**. Although many (perhaps most) uses of Swing will *not* involve drawing directly to the surface of a component, it is available for those applications that need this capability. To write output directly to the surface of a component, you will use one or more drawing methods defined by the AWT, such as **drawLine()** or **drawRect()**. Thus, most of the techniques and methods described in Chapter 25 also apply to Swing. However, there are also some very important differences, and the process is discussed in detail in this section.

Painting Fundamentals

Swing's approach to painting is built on the original AWT-based mechanism, but Swing's implementation offers more finely grained control. Before examining the specifics of Swing-based painting, it is useful to review the AWT-based mechanism that underlies it.

The AWT class **Component** defines a method called **paint()** that is used to draw output directly to the surface of a component. For the most part, **paint()** is not called by your program. (In fact, only in the most unusual cases should it ever be called by your program.) Rather, **paint()** is called by the run-time system whenever a component must be rendered. This situation can occur for several reasons. For example, the window in which the component is displayed can be overwritten by another window and then uncovered. Or, the window might be minimized and then restored. The **paint()** method is also called when a program begins running. When writing AWT-based code, an application will override **paint()** when it needs to write output directly to the surface of the component.

Because **JComponent** inherits **Component**, all Swing's lightweight components inherit the **paint()** method. However, you *will not* override it to paint directly to the surface of a component. The reason is that Swing uses a bit more sophisticated approach to painting that involves three distinct methods: **paintComponent()**, **paintBorder()**, and **paintChildren()**. These methods paint the indicated portion of a component and divide the painting process into its three distinct, logical actions. In a lightweight component, the original AWT method **paint()** simply executes calls to these methods, in the order just shown.

To paint to the surface of a Swing component, you will create a subclass of the component and then override its **paintComponent()** method. This is the method that paints the interior of the component. You will not normally override the other two painting methods. When overriding **paintComponent()**, the first thing you must do is call **super.paintComponent()**, so that the superclass portion of the painting process takes place. (The only time this is not required is when you are taking complete, manual control over how a component is displayed.) After that, write the output that you want to display. The **paintComponent()** method is shown here:

```
protected void paintComponent(Graphics g)
```

The parameter *g* is the graphics context to which output is written.

To cause a component to be painted under program control, call `repaint()`. It works in Swing just as it does for the AWT. The `repaint()` method is defined by **Component**. Calling it causes the system to call `paint()` as soon as it is possible to do so. Because painting is a time-consuming operation, this mechanism allows the run-time system to defer painting momentarily until some higher-priority task has completed, for example. Of course, in Swing the call to `paint()` results in a call to `paintComponent()`. Therefore, to output to the surface of a component, your program will store the output until `paintComponent()` is called. Inside the overridden `paintComponent()`, you will draw the stored output.

Compute the Paintable Area

When drawing to the surface of a component, you must be careful to restrict your output to the area that is inside the border. Although Swing automatically clips any output that will exceed the boundaries of a component, it is still possible to paint into the border, which will then get overwritten when the border is drawn. To avoid this, you must compute the *paintable area* of the component. This is the area defined by the current size of the component minus the space used by the border. Therefore, before you paint to a component, you must obtain the width of the border and then adjust your drawing accordingly.

To obtain the border width, call `getInsets()`, shown here:

```
Insets getInsets()
```

This method is defined by **Container** and overridden by **JComponent**. It returns an **Insets** object that contains the dimensions of the border. The inset values can be obtained by using these fields:

```
int top;  
int bottom;  
int left;  
int right;
```

These values are then used to compute the drawing area given the width and the height of the component. You can obtain the width and height of the component by calling `getWidth()` and `getHeight()` on the component. They are shown here:

```
int getWidth()  
int getHeight()
```

By subtracting the value of the insets, you can compute the usable width and height of the component.

A Paint Example

Here is a program that puts into action the preceding discussion. It creates a class called **PaintPanel** that extends **JPanel**. The program then uses an object of that class to display lines whose endpoints have been generated randomly. Sample output is shown in Figure 31-4.

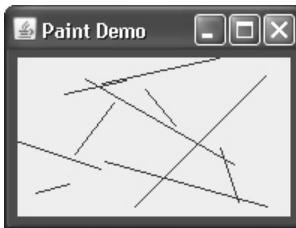


Figure 31-4 Sample output from the **PaintPanel** program

```
// Paint lines to a panel.

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.util.*;

// This class extends JPanel. It overrides
// the paintComponent() method so that random
// lines are plotted in the panel.
class PaintPanel extends JPanel {
    Insets ins; // holds the panel's insets

    Random rand; // used to generate random numbers

    // Construct a panel.
    PaintPanel() {

        // Put a border around the panel.
        setBorder(
            BorderFactory.createLineBorder(Color.RED, 5));

        rand = new Random();
    }

    // Override the paintComponent() method.
    protected void paintComponent(Graphics g) {
        // Always call the superclass method first.
        super.paintComponent(g);

        int x, y, x2, y2;

        // Get the height and width of the component.
        int height = getHeight();
        int width = getWidth();

        // Get the insets.
        ins = getInsets();

        // Draw ten lines whose endpoints are randomly generated.
        for(int i=0; i < 10; i++) {
```

```
// Obtain random coordinates that define
// the endpoints of each line.
x = rand.nextInt(width-ins.left);
y = rand.nextInt(height-ins.bottom);
x2 = rand.nextInt(width-ins.left);
y2 = rand.nextInt(height-ins.bottom);

// Draw the line.
g.drawLine(x, y, x2, y2);
}
}
}

// Demonstrate painting directly onto a panel.
class PaintDemo {

    JLabel jlab;
    PaintPanel pp;

    PaintDemo() {

        // Create a new JFrame container.
        JFrame jfrm = new JFrame("Paint Demo");

        // Give the frame an initial size.
        jfrm.setSize(200, 150);

        // Terminate the program when the user closes the application.
        jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        // Create the panel that will be painted.
        pp = new PaintPanel();

        // Add the panel to the content pane. Because the default
        // border layout is used, the panel will automatically be
        // sized to fit the center region.
        jfrm.add(pp);

        // Display the frame.
        jfrm.setVisible(true);
    }

    public static void main(String args[]) {
        // Create the frame on the event dispatching thread.
        SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                new PaintDemo();
            }
        });
    }
}
```

Let's examine this program closely. The **PaintPanel** class extends **JPanel**. **JPanel** is one of Swing's lightweight containers, which means that it is a component that can be added to the content pane of a **JFrame**. To handle painting, **PaintPanel** overrides the **paintComponent()** method. This enables **PaintPanel** to write directly to the surface of the component when painting takes place. The size of the panel is not specified because the program uses the default border layout and the panel is added to the center. This results in the panel being sized to fill the center. If you change the size of the window, the size of the panel will be adjusted accordingly.

Notice that the constructor also specifies a 5-pixel wide, red border. This is accomplished by setting the border by using the **setBorder()** method, shown here:

```
void setBorder(Border border)
```

Border is the Swing interface that encapsulates a border. You can obtain a border by calling one of the factory methods defined by the **BorderFactory** class. The one used in the program is **createLineBorder()**, which creates a simple line border. It is shown here:

```
static Border createLineBorder(Color clr, int width)
```

Here, *clr* specifies the color of the border and *width* specifies its width in pixels.

Inside the override of **paintComponent()**, notice that it first calls **super.paintComponent()**. As explained, this is necessary to ensure that the component is properly drawn. Next, the width and height of the panel are obtained along with the insets. These values are used to ensure the lines lie within the drawing area of the panel. The drawing area is the overall width and height of a component less the border width. The computations are designed to work with differently sized **PaintPanels** and borders. To prove this, try changing the size of the window. The lines will still all lie within the borders of the panel.

The **PaintDemo** class creates a **PaintPanel** and then adds the panel to the content pane. When the application is first displayed, the overridden **paintComponent()** method is called, and the lines are drawn. Each time you resize or hide and restore the window, a new set of lines are drawn. In all cases, the lines fall within the paintable area.

CHAPTER

32

Exploring Swing

The previous chapter described several of the core concepts relating to Swing and showed the general form of both a Swing application and a Swing applet. This chapter continues the discussion of Swing by presenting an overview of several Swing components, such as buttons, check boxes, trees, and tables. The Swing components provide rich functionality and allow a high level of customization. Because of space limitations, it is not possible to describe all of their features and attributes. Rather, the purpose of this overview is to give you a feel for the capabilities of the Swing component set.

The Swing component classes described in this chapter are shown here:

JButton	JCheckBox	JComboBox	JLabel
JList	JRadioButton	JScrollPane	JTabbedPane
JTable	JTextField	JToggleButton	JTree

These components are all lightweight, which means that they are all derived from **JComponent**.

Also discussed is the **ButtonGroup** class, which encapsulates a mutually exclusive set of Swing buttons, and **ImageIcon**, which encapsulates a graphics image. Both are defined by Swing and packaged in **javax.swing**.

One other point: The Swing components are demonstrated in applets because the code for an applet is more compact than it is for a desktop application. However, the same techniques apply to both applets and applications.

JLabel and ImageIcon

JLabel is Swing's easiest-to-use component. It creates a label and was introduced in the preceding chapter. Here, we will look at **JLabel** a bit more closely. **JLabel** can be used to display text and/or an icon. It is a passive component in that it does not respond to user input. **JLabel** defines several constructors. Here are three of them:

```
JLabel(Icon icon)  
JLabel(String str)  
JLabel(String str, Icon icon, int align)
```

Here, *str* and *icon* are the text and icon used for the label. The *align* argument specifies the horizontal alignment of the text and/or icon within the dimensions of the label. It must be one of the following values: **LEFT**, **RIGHT**, **CENTER**, **LEADING**, or **TRAILING**. These constants are defined in the **SwingConstants** interface, along with several others used by the Swing classes.

Notice that icons are specified by objects of type **Icon**, which is an interface defined by Swing. The easiest way to obtain an icon is to use the **ImageIcon** class. **ImageIcon** implements **Icon** and encapsulates an image. Thus, an object of type **ImageIcon** can be passed as an argument to the **Icon** parameter of **JLabel**'s constructor. There are several ways to provide the image, including reading it from a file or downloading it from a URL. Here is the **ImageIcon** constructor used by the example in this section:

```
ImageIcon(String filename)
```

It obtains the image in the file named *filename*.

The icon and text associated with the label can be obtained by the following methods:

```
Icon getIcon()
String getText()
```

The icon and text associated with a label can be set by these methods:

```
void setIcon(Icon icon)
void setText(String str)
```

Here, *icon* and *str* are the icon and text, respectively. Therefore, using **setText()** it is possible to change the text inside a label during program execution.

The following applet illustrates how to create and display a label containing both an icon and a string. It begins by creating an **ImageIcon** object for the file **hourglass.png**, which depicts an hourglass. This is used as the second argument to the **JLabel** constructor. The first and last arguments for the **JLabel** constructor are the label text and the alignment. Finally, the label is added to the content pane.

```
// Demonstrate JLabel and ImageIcon.
import java.awt.*;
import javax.swing.*;
/*
<applet code="JLabelDemo" width=250 height=200>
</applet>
*/
public class JLabelDemo extends JApplet {
    public void init() {
        try {
            SwingUtilities.invokeAndWait(
                new Runnable() {
                    public void run() {
                        makeGUI();
                    }
                }
            );
        }
    }
}
```

```

        } catch (Exception exc) {
            System.out.println("Can't create because of " + exc);
        }
    }

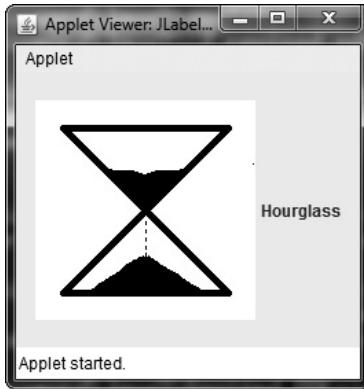
private void makeGUI() {
    // Create an icon.
    ImageIcon ii = new ImageIcon("hourglass.png");

    // Create a label.
    JLabel jl = new JLabel("Hourglass", ii, JLabel.CENTER);

    // Add the label to the content pane.
    add(jl);
}
}

```

Output from the label example is shown here:



JTextField

JTextField is the simplest Swing text component. It is also probably its most widely used text component. **JTextField** allows you to edit one line of text. It is derived from **JTextComponent**, which provides the basic functionality common to Swing text components. **JTextField** uses the **Document** interface for its model. Three of **JTextField**'s constructors are shown here:

```

JTextField(int cols)
JTextField(String str, int cols)
JTextField(String str)

```

Here, *str* is the string to be initially presented, and *cols* is the number of columns in the text field. If no string is specified, the text field is initially empty. If the number of columns is not specified, the text field is sized to fit the specified string.

JTextField generates events in response to user interaction. For example, an **ActionEvent** is fired when the user presses ENTER. A **CaretEvent** is fired each time the caret (i.e., the cursor) changes position. (**CaretEvent** is packaged in **javax.swing.event**.) Other events are

also possible. In many cases, your program will not need to handle these events. Instead, you will simply obtain the string currently in the text field when it is needed. To obtain the text currently in the text field, call `getText()`.

The following example illustrates **JTextField**. It creates a **JTextField** and adds it to the content pane. When the user presses ENTER, an action event is generated. This is handled by displaying the text in the status window.

```
// Demonstrate JTextField.
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
/*
<applet code="JTextFieldDemo" width=300 height=50>
</applet>
*/
public class JTextFieldDemo extends JApplet {
    JTextField jtf;

    public void init() {
        try {
            SwingUtilities.invokeAndWait(
                new Runnable() {
                    public void run() {
                        makeGUI();
                    }
                }
            );
        } catch (Exception exc) {
            System.out.println("Can't create because of " + exc);
        }
    }

    private void makeGUI() {
        // Change to flow layout.
        setLayout(new FlowLayout());

        // Add text field to content pane.
        jtf = new JTextField(15);
        add(jtf);
        jtf.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent ae) {
                // Show text when user presses ENTER.
                showStatus(jtf.getText());
            }
        });
    }
}
```

Output from the text field example is shown here:



The Swing Buttons

Swing defines four types of buttons: **JButton**, **JToggleButton**, **JCheckBox**, and **JRadioButton**. All are subclasses of the **AbstractButton** class, which extends **JComponent**. Thus, all buttons share a set of common traits.

AbstractButton contains many methods that allow you to control the behavior of buttons. For example, you can define different icons that are displayed for the button when it is disabled, pressed, or selected. Another icon can be used as a *rollover* icon, which is displayed when the mouse is positioned over a button. The following methods set these icons:

```
void setDisabledIcon(Icon di)
void setPressedIcon(Icon pi)
void setSelectedIcon(Icon si)
void setRolloverIcon(Icon ri)
```

Here, *di*, *pi*, *si*, and *ri* are the icons to be used for the indicated purpose.

The text associated with a button can be read and written via the following methods:

```
String getText()
void setText(String str)
```

Here, *str* is the text to be associated with the button.

The model used by all buttons is defined by the **ButtonModel** interface. A button generates an action event when it is pressed. Other events are possible. Each of the concrete button classes is examined next.

JButton

The **JButton** class provides the functionality of a push button. You have already seen a simple form of it in the preceding chapter. **JButton** allows an icon, a string, or both to be associated with the push button. Three of its constructors are shown here:

```
JButton(Icon icon)
JButton(String str)
JButton(String str, Icon icon)
```

Here, *str* and *icon* are the string and icon used for the button.

When the button is pressed, an **ActionEvent** is generated. Using the **ActionEvent** object passed to the **actionPerformed()** method of the registered **ActionListener**, you can obtain the *action command* string associated with the button. By default, this is the string displayed inside the button. However, you can set the action command by calling **setActionCommand()** on the button. You can obtain the action command by calling **getActionCommand()** on the event object. It is declared like this:

```
String getActionCommand()
```

The action command identifies the button. Thus, when using two or more buttons within the same application, the action command gives you an easy way to determine which button was pressed.

In the preceding chapter, you saw an example of a text-based button. The following demonstrates an icon-based button. It displays four push buttons and a label. Each button

displays an icon that represents a timepiece. When a button is pressed, the name of that timepiece is displayed in the label.

```
// Demonstrate an icon-based JButton.
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
/*
<applet code="JButtonDemo" width=250 height=750>
</applet>
*/
public class JButtonDemo extends JApplet
implements ActionListener {
JLabel jlab;

public void init() {
try {
SwingUtilities.invokeAndWait(
new Runnable() {
public void run() {
makeGUI();
}
});
} catch (Exception exc) {
System.out.println("Can't create because of " + exc);
}
}

private void makeGUI() {
// Change to flow layout.
setLayout(new FlowLayout());

// Add buttons to content pane.
ImageIcon hourglass = new ImageIcon("hourglass.png");
JButton jb = new JButton(hourglass);
jb.setActionCommand("Hourglass");
jb.addActionListener(this);
add(jb);

ImageIcon analog = new ImageIcon("analog.png");
jb = new JButton(analog);
jb.setActionCommand("Analog Clock");
jb.addActionListener(this);
add(jb);

ImageIcon digital = new ImageIcon("digital.png");
jb = new JButton(digital);
jb.setActionCommand("Digital Clock");
jb.addActionListener(this);
add(jb);
}
```

```

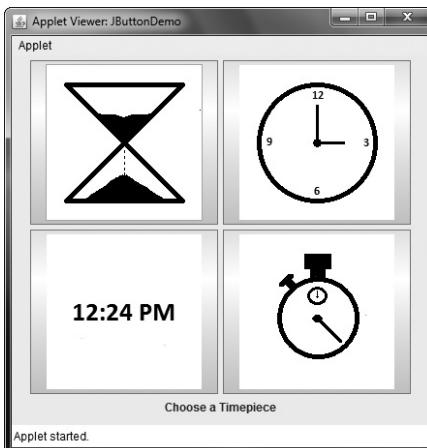
 ImageIcon stopwatch = new ImageIcon("stopwatch.png");
 jb = new JButton(stopwatch);
 jb.setActionCommand("Stopwatch");
 jb.addActionListener(this);
 add(jb);

 // Create and add the label to content pane.
 jlab = new JLabel("Choose a Timepiece");
 add(jlab);
}

// Handle button events.
public void actionPerformed(ActionEvent ae) {
    jlab.setText("You selected " + ae.getActionCommand());
}
}

```

Output from the button example is shown here:



JToggleButton

A useful variation on the push button is called a *toggle button*. A toggle button looks just like a push button, but it acts differently because it has two states: pushed and released. That is, when you press a toggle button, it stays pressed rather than popping back up as a regular push button does. When you press the toggle button a second time, it releases (pops up). Therefore, each time a toggle button is pushed, it toggles between its two states.

Toggle buttons are objects of the **JToggleButton** class. **JToggleButton** implements **AbstractButton**. In addition to creating standard toggle buttons, **JToggleButton** is a superclass for two other Swing components that also represent two-state controls. These are **JCheckBox** and **JRadioButton**, which are described later in this chapter. Thus, **JToggleButton** defines the basic functionality of all two-state components.

JToggleButton defines several constructors. The one used by the example in this section is shown here:

JToggleButton(String str)

This creates a toggle button that contains the text passed in *str*. By default, the button is in the off position. Other constructors enable you to create toggle buttons that contain images, or images and text.

JToggleButton uses a model defined by a nested class called **JToggleButton.ToggleButtonModel**. Normally, you won't need to interact directly with the model to use a standard toggle button.

Like **JButton**, **JToggleButton** generates an action event each time it is pressed. Unlike **JButton**, however, **JToggleButton** also generates an item event. This event is used by those components that support the concept of selection. When a **JToggleButton** is pressed in, it is selected. When it is popped out, it is deselected.

To handle item events, you must implement the **ItemListener** interface. Recall from Chapter 24, that each time an item event is generated, it is passed to the **itemStateChanged()** method defined by **ItemListener**. Inside **itemStateChanged()**, the **getItem()** method can be called on the **ItemEvent** object to obtain a reference to the **JToggleButton** instance that generated the event. It is shown here:

```
Object getItem()
```

A reference to the button is returned. You will need to cast this reference to **JToggleButton**.

The easiest way to determine a toggle button's state is by calling the **isSelected()** method (inherited from **AbstractButton**) on the button that generated the event. It is shown here:

```
boolean isSelected()
```

It returns **true** if the button is selected and **false** otherwise.

Here is an example that uses a toggle button. Notice how the item listener works. It simply calls **isSelected()** to determine the button's state.

```
// Demonstrate JToggleButton.
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
/*
<applet code="JToggleButtonDemo" width=200 height=80>
</applet>
*/
public class JToggleButtonDemo extends JApplet {
    JLabel jlab;
    JToggleButton jtbn;
    public void init() {
        try {
            SwingUtilities.invokeAndWait(
                new Runnable() {
                    public void run() {
                        makeGUI();
                    }
                }
            );
        }
    }
}
```

```

        } catch (Exception exc) {
            System.out.println("Can't create because of " + exc);
        }
    }

private void makeGUI() {
    // Change to flow layout.
    setLayout(new FlowLayout());

    // Create a label.
    jlab = new JLabel("Button is off.");

    // Make a toggle button.
    jbtn = new JToggleButton("On/Off");

    // Add an item listener for the toggle button.
    jbtn.addItemListener(new ItemListener() {
        public void itemStateChanged(ItemEvent ie) {
            if(jbtn.isSelected())
                jlab.setText("Button is on.");
            else
                jlab.setText("Button is off.");
        }
    });

    // Add the toggle button and label to the content pane.
    add(jbtn);
    add(jlab);
}
}

```

The output from the toggle button example is shown here:



Check Boxes

The **JCheckBox** class provides the functionality of a check box. Its immediate superclass is **JToggleButton**, which provides support for two-state buttons, as just described. **JCheckBox** defines several constructors. The one used here is

`JCheckBox(String str)`

It creates a check box that has the text specified by *str* as a label. Other constructors let you specify the initial selection state of the button and specify an icon.

When the user selects or deselects a check box, an **ItemEvent** is generated. You can obtain a reference to the **JCheckBox** that generated the event by calling **getItem()** on the **ItemEvent** passed to the **itemStateChanged()** method defined by **ItemListener**. The easiest way to determine the selected state of a check box is to call **isSelected()** on the **JCheckBox** instance.

The following example illustrates check boxes. It displays four check boxes and a label. When the user clicks a check box, an **ItemEvent** is generated. Inside the **itemStateChanged()** method, **getItem()** is called to obtain a reference to the **JCheckBox** object that generated the event. Next, a call to **isSelected()** determines if the box was selected or cleared. The **getText()** method gets the text for that check box and uses it to set the text inside the label.

```
// Demonstrate JCheckbox.
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
/*
<applet code="JCheckBoxDemo" width=270 height=50>
</applet>
*/
public class JCheckBoxDemo extends JApplet
implements ItemListener {
    JLabel jlab;

    public void init() {
        try {
            SwingUtilities.invokeAndWait(
                new Runnable() {
                    public void run() {
                        makeGUI();
                    }
                }
            );
        } catch (Exception exc) {
            System.out.println("Can't create because of " + exc);
        }
    }

    private void makeGUI() {

        // Change to flow layout.
       setLayout(new FlowLayout());

        // Add check boxes to the content pane.
        JCheckBox cb = new JCheckBox("C");
        cb.addItemListener(this);
        add(cb);

        cb = new JCheckBox("C++");
        cb.addItemListener(this);
        add(cb);
    }
}
```

```

cb = new JCheckBox("Java");
cb.addItemListener(this);
add(cb);

cb = new JCheckBox("Perl");
cb.addItemListener(this);
add(cb);

// Create the label and add it to the content pane.
jlab = new JLabel("Select languages");
add(jlab);
}

// Handle item events for the check boxes.
public void itemStateChanged(ItemEvent ie) {
    JCheckBox cb = (JCheckBox)ie.getItem();

    if(cb.isSelected())
        jlab.setText(cb.getText() + " is selected");
    else
        jlab.setText(cb.getText() + " is cleared");
}
}

```

Output from this example is shown here:



Radio Buttons

Radio buttons are a group of mutually exclusive buttons, in which only one button can be selected at any one time. They are supported by the **JRadioButton** class, which extends **JToggleButton**. **JRadioButton** provides several constructors. The one used in the example is shown here:

```
JRadioButton(String str)
```

Here, *str* is the label for the button. Other constructors let you specify the initial selection state of the button and specify an icon.

In order for their mutually exclusive nature to be activated, radio buttons must be configured into a group. Only one of the buttons in the group can be selected at any time. For example, if a user presses a radio button that is in a group, any previously selected button in that group is automatically deselected. A button group is created by the **ButtonGroup** class. Its default constructor is invoked for this purpose. Elements are then added to the button group via the following method:

```
void add(AbstractButton ab)
```

Here, *ab* is a reference to the button to be added to the group.

A **JRadioButton** generates action events, item events, and change events each time the button selection changes. Most often, it is the action event that is handled, which means that you will normally implement the **ActionListener** interface. Recall that the only method defined by **ActionListener** is **actionPerformed()**. Inside this method, you can use a number of different ways to determine which button was selected. First, you can check its action command by calling **getActionCommand()**. By default, the action command is the same

as the button label, but you can set the action command to something else by calling `setActionCommand()` on the radio button. Second, you can call `getSource()` on the `ActionEvent` object and check that reference against the buttons. Third, you can check each radio button to find out which one is currently selected by calling `isSelected()` on each button. Finally, each button could use its own action event handler implemented as either an anonymous inner class or a lambda expression. Remember, each time an action event occurs, it means that the button being selected has changed and that one and only one button will be selected.

The following example illustrates how to use radio buttons. Three radio buttons are created. The buttons are then added to a button group. As explained, this is necessary to cause their mutually exclusive behavior. Pressing a radio button generates an action event, which is handled by `actionPerformed()`. Within that handler, the `getActionCommand()` method gets the text that is associated with the radio button and uses it to set the text within a label.

```
// Demonstrate JRadioButton
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
/*
<applet code="JRadioButtonDemo" width=300 height=50>
</applet>
*/
public class JRadioButtonDemo extends JApplet
implements ActionListener {
    JLabel jlab;

    public void init() {
        try {
            SwingUtilities.invokeAndWait(
                new Runnable() {
                    public void run() {
                        makeGUI();
                    }
                }
            );
        } catch (Exception exc) {
            System.out.println("Can't create because of " + exc);
        }
    }

    private void makeGUI() {
        // Change to flow layout.
       setLayout(new FlowLayout());

        // Create radio buttons and add them to content pane.
        JRadioButton b1 = new JRadioButton("A");
        b1.addActionListener(this);
        add(b1);
```

```

JRadioButton b2 = new JRadioButton("B");
b2.addActionListener(this);
add(b2);

JRadioButton b3 = new JRadioButton("C");
b3.addActionListener(this);
add(b3);

// Define a button group.
ButtonGroup bg = new ButtonGroup();
bg.add(b1);
bg.add(b2);
bg.add(b3);

// Create a label and add it to the content pane.
jlab = new JLabel("Select One");
add(jlab);
}

// Handle button selection.
public void actionPerformed(ActionEvent ae) {
    jlab.setText("You selected " + ae.getActionCommand());
}
}

```

Output from the radio button example is shown here:



JTabbedPane

JTabbedPane encapsulates a *tabbed pane*. It manages a set of components by linking them with tabs. Selecting a tab causes the component associated with that tab to come to the forefront. Tabbed panes are very common in the modern GUI, and you have no doubt used them many times. Given the complex nature of a tabbed pane, they are surprisingly easy to create and use.

JTabbedPane defines three constructors. We will use its default constructor, which creates an empty control with the tabs positioned across the top of the pane. The other two constructors let you specify the location of the tabs, which can be along any of the four sides. **JTabbedPane** uses the **SingleSelectionModel** model.

Tabs are added by calling **addTab()**. Here is one of its forms:

```
void addTab(String name, Component comp)
```

Here, *name* is the name for the tab, and *comp* is the component that should be added to the tab. Often, the component added to a tab is a **JPanel** that contains a group of related components. This technique allows a tab to hold a set of components.

The general procedure to use a tabbed pane is outlined here:

1. Create an instance of **JTabbedPane**.
2. Add each tab by calling **addTab()**.
3. Add the tabbed pane to the content pane.

The following example illustrates a tabbed pane. The first tab is titled "Cities" and contains four buttons. Each button displays the name of a city. The second tab is titled "Colors" and contains three check boxes. Each check box displays the name of a color. The third tab is titled "Flavors" and contains one combo box. This enables the user to select one of three flavors.

```
// Demonstrate JTabbedPane.
import javax.swing.*;
/*
<applet code="JTabbedPaneDemo" width=400 height=100>
</applet>
*/
public class JTabbedPaneDemo extends JApplet {

    public void init() {
        try {
            SwingUtilities.invokeAndWait(
                new Runnable() {
                    public void run() {
                        makeGUI();
                    }
                }
            );
        } catch (Exception exc) {
            System.out.println("Can't create because of " + exc);
        }
    }

    private void makeGUI() {

        JTabbedPane jtp = new JTabbedPane();
        jtp.addTab("Cities", new CitiesPanel());
        jtp.addTab("Colors", new ColorsPanel());
        jtp.addTab("Flavors", new FlavorsPanel());
        add(jtp);
    }
}

// Make the panels that will be added to the tabbed pane.
class CitiesPanel extends JPanel {

    public CitiesPanel() {
        JButton b1 = new JButton("New York");
        add(b1);
        JButton b2 = new JButton("London");
        add(b2);
    }
}
```

```
        JButton b3 = new JButton("Hong Kong");
        add(b3);
        JButton b4 = new JButton("Tokyo");
        add(b4);
    }
}

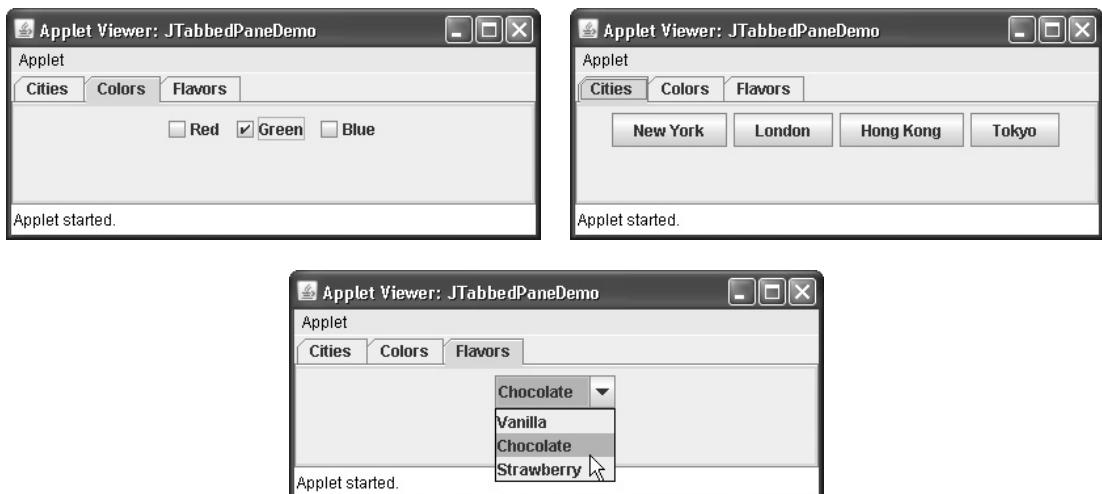
class ColorsPanel extends JPanel {

    public ColorsPanel() {
        JCheckBox cb1 = new JCheckBox("Red");
        add(cb1);
        JCheckBox cb2 = new JCheckBox("Green");
        add(cb2);
        JCheckBox cb3 = new JCheckBox("Blue");
        add(cb3);
    }
}

class FlavorsPanel extends JPanel {

    public FlavorsPanel() {
        JComboBox<String> jcb = new JComboBox<String>();
        jcb.addItem("Vanilla");
        jcb.addItem("Chocolate");
        jcb.addItem("Strawberry");
        add(jcb);
    }
}
```

Output from the tabbed pane example is shown in the following three illustrations:



JScrollPane

JScrollPane is a lightweight container that automatically handles the scrolling of another component. The component being scrolled can be either an individual component, such as a table, or a group of components contained within another lightweight container, such as a **JPanel**. In either case, if the object being scrolled is larger than the viewable area, horizontal and/or vertical scroll bars are automatically provided, and the component can be scrolled through the pane. Because **JScrollPane** automates scrolling, it usually eliminates the need to manage individual scroll bars.

The viewable area of a scroll pane is called the *viewport*. It is a window in which the component being scrolled is displayed. Thus, the viewport displays the visible portion of the component being scrolled. The scroll bars scroll the component through the viewport. In its default behavior, a **JScrollPane** will dynamically add or remove a scroll bar as needed. For example, if the component is taller than the viewport, a vertical scroll bar is added. If the component will completely fit within the viewport, the scroll bars are removed.

JScrollPane defines several constructors. The one used in this chapter is shown here:

`JScrollPane(Component comp)`

The component to be scrolled is specified by *comp*. Scroll bars are automatically displayed when the content of the pane exceeds the dimensions of the viewport.

Here are the steps to follow to use a scroll pane:

1. Create the component to be scrolled.
2. Create an instance of **JScrollPane**, passing to it the object to scroll.
3. Add the scroll pane to the content pane.

The following example illustrates a scroll pane. First, a **JPanel** object is created, and 400 buttons are added to it, arranged into 20 columns. This panel is then added to a scroll pane, and the scroll pane is added to the content pane. Because the panel is larger than the viewport, vertical and horizontal scroll bars appear automatically. You can use the scroll bars to scroll the buttons into view.

```
// Demonstrate JScrollPane.
import java.awt.*;
import javax.swing.*;
/*
<applet code="JScrollPaneDemo" width=300 height=250>
</applet>
*/
public class JScrollPaneDemo extends JApplet {
    public void init() {
        try {
            SwingUtilities.invokeAndWait(
                new Runnable() {
                    public void run() {

```

```
        makeGUI () ;
    }
}
);
} catch (Exception exc) {
    System.out.println("Can't create because of " + exc);
}
}

private void makeGUI() {
    // Add 400 buttons to a panel.
    JPanel jp = new JPanel();
    jp.setLayout(new GridLayout(20, 20));
    int b = 0;

    for(int i = 0; i < 20; i++) {
        for(int j = 0; j < 20; j++) {
            jp.add(new JButton("Button " + b));
            ++b;
        }
    }

    // Create the scroll pane.
    JScrollPane jsp = new JScrollPane(jp);

    // Add the scroll pane to the content pane.
    // Because the default border layout is used,
    // the scroll pane will be added to the center.
    add(jsp, BorderLayout.CENTER);
}
}
```

Output from the scroll pane example is shown here:



JList

In Swing, the basic list class is called **JList**. It supports the selection of one or more items from a list. Although the list often consists of strings, it is possible to create a list of just about any object that can be displayed. **JList** is so widely used in Java that it is highly unlikely that you have not seen one before.

In the past, the items in a **JList** were represented as **Object** references. However, beginning with JDK 7, **JList** was made generic and is now declared like this:

```
class JList<E>
```

Here, **E** represents the type of the items in the list.

JList provides several constructors. The one used here is

```
JList(E[ ] items)
```

This creates a **JList** that contains the items in the array specified by *items*.

JList is based on two models. The first is **ListModel**. This interface defines how access to the list data is achieved. The second model is the **ListSelectionModel** interface, which defines methods that determine what list item or items are selected.

Although a **JList** will work properly by itself, most of the time you will wrap a **JList** inside a **JScrollPane**. This way, long lists will automatically be scrollable, which simplifies GUI design. It also makes it easy to change the number of entries in a list without having to change the size of the **JList** component.

A **JList** generates a **ListSelectionEvent** when the user makes or changes a selection. This event is also generated when the user deselects an item. It is handled by implementing **ListSelectionListener**. This listener specifies only one method, called **valueChanged()**, which is shown here:

```
void valueChanged(ListSelectionEvent le)
```

Here, *le* is a reference to the event. Although **ListSelectionEvent** does provide some methods of its own, normally you will interrogate the **JList** object itself to determine what has occurred. Both **ListSelectionEvent** and **ListSelectionListener** are packaged in **javax.swing.event**.

By default, a **JList** allows the user to select multiple ranges of items within the list, but you can change this behavior by calling **setSelectionMode()**, which is defined by **JList**. It is shown here:

```
void setSelectionMode(int mode)
```

Here, *mode* specifies the selection mode. It must be one of these values defined by **ListSelectionModel**:

SINGLE_SELECTION

SINGLE_INTERVAL_SELECTION

MULTIPLE_INTERVAL_SELECTION

The default, multiple-interval selection, lets the user select multiple ranges of items within a list. With single-interval selection, the user can select one range of items. With single

selection, the user can select only a single item. Of course, a single item can be selected in the other two modes, too. It's just that they also allow a range to be selected.

You can obtain the index of the first item selected, which will also be the index of the only selected item when using single-selection mode, by calling **getSelectedIndex()**, shown here:

```
int getSelectedIndex()
```

Indexing begins at zero. So, if the first item is selected, this method will return 0. If no item is selected, -1 is returned.

Instead of obtaining the index of a selection, you can obtain the value associated with the selection by calling **getSelectedValue()**:

```
E getSelectedValue()
```

It returns a reference to the first selected value. If no value has been selected, it returns **null**.

The following applet demonstrates a simple **JList**, which holds a list of cities. Each time a city is selected in the list, a **ListSelectionEvent** is generated, which is handled by the **valueChanged()** method defined by **ListSelectionListener**. It responds by obtaining the index of the selected item and displaying the name of the selected city in a label.

```
// Demonstrate JList.
import javax.swing.*;
import javax.swing.event.*;
import java.awt.*;
import java.awt.event.*;

/*
<applet code="JListDemo" width=200 height=120>
</applet>
*/

public class JListDemo extends JApplet {
    JList<String> jlst;
    JLabel jlab;
    JScrollPane jscrlp;

    // Create an array of cities.
    String Cities[] = { "New York", "Chicago", "Houston",
                        "Denver", "Los Angeles", "Seattle",
                        "London", "Paris", "New Delhi",
                        "Hong Kong", "Tokyo", "Sydney" };

    public void init() {
        try {
            SwingUtilities.invokeAndWait(
                new Runnable() {
                    public void run() {
                        makeGUI();
                    }
                }
            );
        } catch (Exception exc) {
            System.out.println("Can't create because of " + exc);
        }
    }
}
```

```
        }
    }

    private void makeGUI() {
        // Change to flow layout.
        setLayout(new FlowLayout());

        // Create a JList.
        jlst = new JList<String>(Cities);

        // Set the list selection mode to single selection.
        jlst.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);

        // Add the list to a scroll pane.
        jscrlp = new JScrollPane(jlst);

        // Set the preferred size of the scroll pane.
        jscrlp.setPreferredSize(new Dimension(120, 90));

        // Make a label that displays the selection.
        jlab = new JLabel("Choose a City");

        // Add selection listener for the list.
        jlst.addListSelectionListener(new ListSelectionListener() {
            public void valueChanged(ListSelectionEvent le) {
                // Get the index of the changed item.
                int idx = jlst.getSelectedIndex();

                // Display selection, if item was selected.
                if(idx != -1)
                    jlab.setText("Current selection: " + Cities[idx]);
                else // Otherwise, reprompt.
                    jlab.setText("Choose a City");
            }
        });

        // Add the list and label to the content pane.
        add(jscrlp);
        add(jlab);
    }
}
```

Output from the list example is shown here:



JComboBox

Swing provides a *combo box* (a combination of a text field and a drop-down list) through the **JComboBox** class. A combo box normally displays one entry, but it will also display a drop-down list that allows a user to select a different entry. You can also create a combo box that lets the user enter a selection into the text field.

In the past, the items in a **JComboBox** were represented as **Object** references. However, beginning with JDK 7, **JComboBox** was made generic and is now declared like this:

```
class JComboBox<E>
```

Here, **E** represents the type of the items in the combo box.

The **JComboBox** constructor used by the example is shown here:

```
JComboBox(E[ ] items)
```

Here, *items* is an array that initializes the combo box. Other constructors are available.

JComboBox uses the **ComboBoxModel**. Mutable combo boxes (those whose entries can be changed) use the **MutableComboBoxModel**.

In addition to passing an array of items to be displayed in the drop-down list, items can be dynamically added to the list of choices via the **addItem()** method, shown here:

```
void addItem(E obj)
```

Here, *obj* is the object to be added to the combo box. This method must be used only with mutable combo boxes.

JComboBox generates an action event when the user selects an item from the list. **JComboBox** also generates an item event when the state of selection changes, which occurs when an item is selected or deselected. Thus, changing a selection means that two item events will occur: one for the deselected item and another for the selected item. Often, it is sufficient to simply listen for action events, but both event types are available for your use.

One way to obtain the item selected in the list is to call **getSelectedItem()** on the combo box. It is shown here:

```
Object getSelectedItem()
```

You will need to cast the returned value into the type of object stored in the list.

The following example demonstrates the combo box. The combo box contains entries for "Hourglass", "Analog", "Digital", and "Stopwatch". When a timepiece is selected, an icon-based label is updated to display it. You can see how little code is required to use this powerful component.

```
// Demonstrate JComboBox.
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
/*
<applet code="JComboBoxDemo" width=300 height=200>
</applet>
*/
```

```
public class JComboBoxDemo extends JApplet {
    JLabel jlab;
    ImageIcon hourglass, analog, digital, stopwatch;
    JComboBox<String> jcb;

    String timepieces[] = { "Hourglass", "Analog", "Digital", "Stopwatch" };

    public void init() {
        try {
            SwingUtilities.invokeAndWait(
                new Runnable() {
                    public void run() {
                        makeGUI();
                    }
                }
            );
        } catch (Exception exc) {
            System.out.println("Can't create because of " + exc);
        }
    }

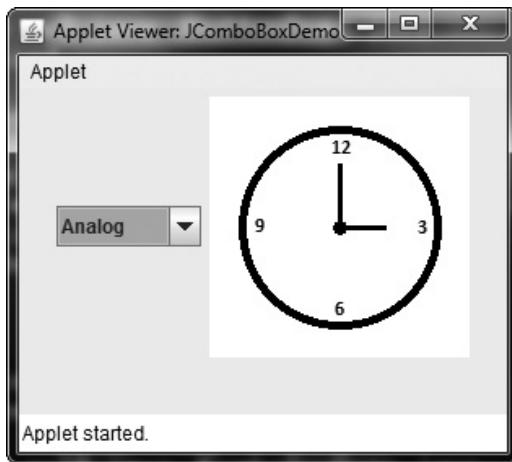
    private void makeGUI() {
        // Change to flow layout.
        setLayout(new FlowLayout());

        // Instantiate a combo box and add it to the content pane.
        jcb = new JComboBox<String>(timepieces);
        add(jcb);

        // Handle selections.
        jcb.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent ae) {
                String s = (String) jcb.getSelectedItem();
                jlab.setIcon(new ImageIcon(s + ".png"));
            }
        });

        // Create a label and add it to the content pane.
        jlab = new JLabel(new ImageIcon("hourglass.png"));
        add(jlab);
    }
}
```

Output from the combo box example is shown here:



Trees

A *tree* is a component that presents a hierarchical view of data. The user has the ability to expand or collapse individual subtrees in this display. Trees are implemented in Swing by the **JTree** class. A sampling of its constructors is shown here:

```
JTree(Object obj[])
JTree(Vector<?> v)
JTree(TreeNode tn)
```

In the first form, the tree is constructed from the elements in the array *obj*. The second form constructs the tree from the elements of vector *v*. In the third form, the tree whose root node is specified by *tn* specifies the tree.

Although **JTree** is packaged in **javax.swing**, its support classes and interfaces are packaged in **javax.swing.tree**. This is because the number of classes and interfaces needed to support **JTree** is quite large.

JTree relies on two models: **TreeModel** and **TreeSelectionModel**. A **JTree** generates a variety of events, but three relate specifically to trees: **TreeExpansionEvent**, **TreeSelectionEvent**, and **TreeModelEvent**. **TreeExpansionEvent** events occur when a node is expanded or collapsed. A **TreeSelectionEvent** is generated when the user selects or deselects a node within the tree. A **TreeModelEvent** is fired when the data or structure of the tree changes. The listeners for these events are **TreeExpansionListener**, **TreeSelectionListener**, and **TreeModelListener**, respectively. The tree event classes and listener interfaces are packaged in **javax.swing.event**.

The event handled by the sample program shown in this section is **TreeSelectionEvent**. To listen for this event, implement **TreeSelectionListener**. It defines only one method, called **valueChanged()**, which receives the **TreeSelectionEvent** object. You can obtain the path to the selected object by calling **getPath()**, shown here, on the event object:

```
TreePath getPath()
```

It returns a **TreePath** object that describes the path to the changed node. The **TreePath** class encapsulates information about a path to a particular node in a tree. It provides several constructors and methods. In this book, only the **toString()** method is used. It returns a string that describes the path.

The **TreeNode** interface declares methods that obtain information about a tree node. For example, it is possible to obtain a reference to the parent node or an enumeration of the child nodes. The **MutableTreeNode** interface extends **TreeNode**. It declares methods that can insert and remove child nodes or change the parent node.

The **DefaultMutableTreeNode** class implements the **MutableTreeNode** interface. It represents a node in a tree. One of its constructors is shown here:

```
DefaultMutableTreeNode(Object obj)
```

Here, *obj* is the object to be enclosed in this tree node. The new tree node doesn't have a parent or children.

To create a hierarchy of tree nodes, the **add()** method of **DefaultMutableTreeNode** can be used. Its signature is shown here:

```
void add(MutableTreeNode child)
```

Here, *child* is a mutable tree node that is to be added as a child to the current node.

JTree does not provide any scrolling capabilities of its own. Instead, a **JTree** is typically placed within a **JScrollPane**. This way, a large tree can be scrolled through a smaller viewport.

Here are the steps to follow to use a tree:

1. Create an instance of **JTree**.
2. Create a **JScrollPane** and specify the tree as the object to be scrolled.
3. Add the tree to the scroll pane.
4. Add the scroll pane to the content pane.

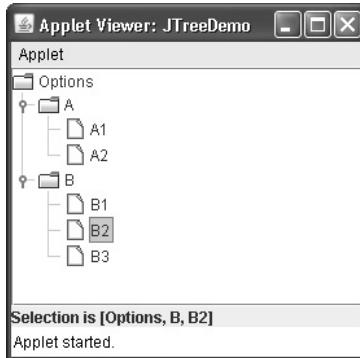
The following example illustrates how to create a tree and handle selections. The program creates a **DefaultMutableTreeNode** instance labeled "Options". This is the top node of the tree hierarchy. Additional tree nodes are then created, and the **add()** method is called to connect these nodes to the tree. A reference to the top node in the tree is provided as the argument to the **JTree** constructor. The tree is then provided as the argument to the **JScrollPane** constructor. This scroll pane is then added to the content pane. Next, a label is created and added to the content pane. The tree selection is displayed in this label. To receive selection events from the tree, a **TreeSelectionListener** is registered for the tree. Inside the **valueChanged()** method, the path to the current selection is obtained and displayed.

```
// Demonstrate JTree.
import java.awt.*;
import javax.swing.event.*;
import javax.swing.*;
import javax.swing.tree.*;
/*
<applet code="JTreeDemo" width=400 height=200>
</applet>
*/
```

```
public class JTreeDemo extends JApplet {  
    JTree tree;  
    JLabel jlab;  
  
    public void init() {  
        try {  
            SwingUtilities.invokeAndWait(  
                new Runnable() {  
                    public void run() {  
                        makeGUI();  
                    }  
                }  
            );  
        } catch (Exception exc) {  
            System.out.println("Can't create because of " + exc);  
        }  
    }  
  
    private void makeGUI() {  
  
        // Create top node of tree.  
        DefaultMutableTreeNode top = new DefaultMutableTreeNode("Options");  
  
        // Create subtree of "A".  
        DefaultMutableTreeNode a = new DefaultMutableTreeNode("A");  
        top.add(a);  
        DefaultMutableTreeNode a1 = new DefaultMutableTreeNode("A1");  
        a.add(a1);  
        DefaultMutableTreeNode a2 = new DefaultMutableTreeNode("A2");  
        a.add(a2);  
  
        // Create subtree of "B"  
        DefaultMutableTreeNode b = new DefaultMutableTreeNode("B");  
        top.add(b);  
        DefaultMutableTreeNode b1 = new DefaultMutableTreeNode("B1");  
        b.add(b1);  
        DefaultMutableTreeNode b2 = new DefaultMutableTreeNode("B2");  
        b.add(b2);  
        DefaultMutableTreeNode b3 = new DefaultMutableTreeNode("B3");  
        b.add(b3);  
  
        // Create the tree.  
        tree = new JTree(top);  
  
        // Add the tree to a scroll pane.  
        JScrollPane jsp = new JScrollPane(tree);  
  
        // Add the scroll pane to the content pane.  
        add(jsp);  
  
        // Add the label to the content pane.  
        jlab = new JLabel();  
        add(jlab, BorderLayout.SOUTH);  
    }  
}
```

```
// Handle tree selection events.  
tree.addTreeSelectionListener(new TreeSelectionListener() {  
    public void valueChanged(TreeSelectionEvent tse) {  
        jlab.setText("Selection is " + tse.getPath());  
    }  
});  
}  
}
```

Output from the tree example is shown here:



The string presented in the text field describes the path from the top tree node to the selected node.

JTable

JTable is a component that displays rows and columns of data. You can drag the cursor on column boundaries to resize columns. You can also drag a column to a new position. Depending on its configuration, it is also possible to select a row, column, or cell within the table, and to change the data within a cell. **JTable** is a sophisticated component that offers many more options and features than can be discussed here. (It is perhaps Swing's most complicated component.) However, in its default configuration, **JTable** still offers substantial functionality that is easy to use—especially if you simply want to use the table to present data in a tabular format. The brief overview presented here will give you a general understanding of this powerful component.

Like **JTree**, **JTable** has many classes and interfaces associated with it. These are packaged in **javax.swing.table**.

At its core, **JTable** is conceptually simple. It is a component that consists of one or more columns of information. At the top of each column is a heading. In addition to describing the data in a column, the heading also provides the mechanism by which the user can change the size of a column or change the location of a column within the table. **JTable** does not provide any scrolling capabilities of its own. Instead, you will normally wrap a **JTable** inside a **JScrollPane**.

JTable supplies several constructors. The one used here is

[Table(Object *data*[][], Object *colHeads*[])]

Here, `data` is a two-dimensional array of the information to be presented, and `colHeads` is a one-dimensional array with the column headings.

JTable relies on three models. The first is the table model, which is defined by the **TableModel** interface. This model defines those things related to displaying data in a two-dimensional format. The second is the table column model, which is represented by **TableColumnModel**. **JTable** is defined in terms of columns, and it is **TableColumnModel** that specifies the characteristics of a column. These two models are packaged in **javax.swing.table**. The third model determines how items are selected, and it is specified by the **ListSelectionModel**, which was described when **JList** was discussed.

A **JTable** can generate several different events. The two most fundamental to a table's operation are **ListSelectionEvent** and **TableModelEvent**. A **ListSelectionEvent** is generated when the user selects something in the table. By default, **JTable** allows you to select one or more complete rows, but you can change this behavior to allow one or more columns, or one or more individual cells to be selected. A **TableModelEvent** is fired when that table's data changes in some way. Handling these events requires a bit more work than it does to handle the events generated by the previously described components and is beyond the scope of this book. However, if you simply want to use **JTable** to display data (as the following example does), then you don't need to handle any events.

Here are the steps required to set up a simple **JTable** that can be used to display data:

1. Create an instance of **JTable**.
2. Create a **JScrollPane** object, specifying the table as the object to scroll.
3. Add the table to the scroll pane.
4. Add the scroll pane to the content pane.

The following example illustrates how to create and use a simple table. A one-dimensional array of strings called `colHeads` is created for the column headings. A two-dimensional array of strings called `data` is created for the table cells. You can see that each element in the array is an array of three strings. These arrays are passed to the **JTable** constructor. The table is added to a scroll pane, and then the scroll pane is added to the content pane. The table displays the data in the `data` array. The default table configuration also allows the contents of a cell to be edited. Changes affect the underlying array, which is `data` in this case.

```
// Demonstrate JTable.
import java.awt.*;
import javax.swing.*;
/*
<applet code="JTableDemo" width=400 height=200>
</applet>
*/
public class JTableDemo extends JApplet {
    public void init() {
        try {
            SwingUtilities.invokeAndWait(
                new Runnable() {
                    public void run() {
                        makeGUI();
                    }
                }
            );
        } catch (Exception e) {
        }
    }
    void makeGUI() {
        // Create the table.
        JTable table = new JTable(data, colHeads);
        // Create the scroll pane and add the table to it.
        JScrollPane scrollPane = new JScrollPane(table);
        // Set the scroll pane as the content pane.
        setContentPane(scrollPane);
    }
}
```

```
        }
    }
}
} catch (Exception exc) {
    System.out.println("Can't create because of " + exc);
}
}

private void makeGUI() {

    // Initialize column headings.
    String[] colHeads = { "Name", "Extension", "ID#" };

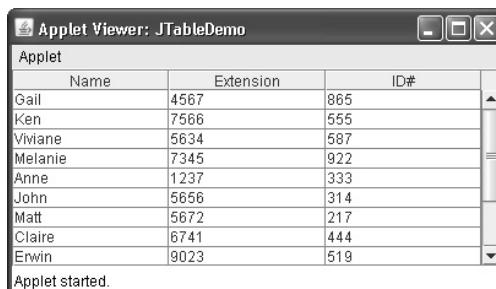
    // Initialize data.
    Object[][] data = {
        { "Gail", "4567", "865" },
        { "Ken", "7566", "555" },
        { "Viviane", "5634", "587" },
        { "Melanie", "7345", "922" },
        { "Anne", "1237", "333" },
        { "John", "5656", "314" },
        { "Matt", "5672", "217" },
        { "Claire", "6741", "444" },
        { "Erwin", "9023", "519" },
        { "Ellen", "1134", "532" },
        { "Jennifer", "5689", "112" },
        { "Ed", "9030", "133" },
        { "Helen", "6751", "145" }
    };

    // Create the table.
    JTable table = new JTable(data, colHeads);

    // Add the table to a scroll pane.
    JScrollPane jsp = new JScrollPane(table);

    // Add the scroll pane to the content pane.
    add(jsp);
}
}
```

Output from this example is shown here:



CHAPTER

33

Introducing Swing Menus

This chapter introduces another fundamental aspect of the Swing GUI environment: the menu. Menus form an integral part of many applications because they present the program's functionality to the user. Because of their importance, Swing provides extensive support for menus. They are an area in which Swing's power is readily apparent.

The Swing menu system supports several key elements, including

- The menu bar, which is the main menu for an application.
- The standard menu, which can contain either items to be selected or other menus (submenus).
- The popup menu, which is usually activated by right-clicking the mouse.
- The toolbar, which provides rapid access to program functionality, often paralleling menu items.
- The action, which enables two or more different components to be managed by a single object. Actions are commonly used with menus and toolbars.

Swing menus also support accelerator keys, which enable menu items to be selected without having to activate the menu, and mnemonics, which allow a menu item to be selected by the keyboard once the menu options are displayed.

Menu Basics

The Swing menu system is supported by a group of related classes. The ones used in this chapter are shown in Table 33-1, and they represent the core of the menu system. Although they may seem a bit confusing at first, Swing menus are quite easy to use. Swing allows a high degree of customization, if desired; however, you will normally use the menu classes as-is because they support all of the most needed options. For example, you can easily add images and keyboard shortcuts to a menu.

Class	Description
JMenuBar	An object that holds the top-level menu for the application.
JMenu	A standard menu. A menu consists of one or more JMenuItem s.
JMenuItem	An object that populates menus.
JCheckBoxMenuItem	A check box menu item.
JRadioButtonMenuItem	A radio button menu item
JSeparator	The visual separator between menu items.
JPopupMenu	A menu that is typically activated by right-clicking the mouse.

Table 33-1 The Core Swing Menu Classes

Here is a brief overview of how the classes fit together. To create the top-level menu for an application, you first create a **JMenuBar** object. This class is, loosely speaking, a container for menus. To the **JMenuBar** instance, you will add instances of **JMenu**. Each **JMenu** object defines a menu. That is, each **JMenu** object contains one or more selectable items. The items displayed by a **JMenu** are objects of **JMenuItem**. Thus, a **JMenuItem** defines a selection that can be chosen by the user.

As an alternative or adjunct to menus that descend from the menu bar, you can also create stand-alone, popup menus. To create a popup menu, first create an object of type **JPopupMenu**. Then, add **JMenuItem**s to it. A popup menu is normally activated by clicking the right mouse button when the mouse is over a component for which a popup menu has been defined.

In addition to “standard” menu items, you can also include check boxes and radio buttons in a menu. A check box menu item is created by **JCheckBoxMenuItem**. A radio button menu item is created by **JRadioButtonMenuItem**. Both of these classes extend **JMenuItem**. They can be used in standard menus and popup menus.

JToolBar creates a stand-alone component that is related to the menu. It is often used to provide fast access to functionality contained within the menus of the application. For example, a toolbar might provide fast access to the formatting commands supported by a word processor.

JSeparator is a convenience class that creates a separator line in a menu.

One key point to understand about Swing menus is that each menu item extends **AbstractButton**. Recall that **AbstractButton** is also the superclass of all of Swing’s button components, such as **JButton**. Thus, all menu items are, essentially, buttons. Obviously, they won’t actually look like buttons when used in a menu, but they will, in many ways, act like buttons. For example, selecting a menu item generates an action event in the same way that pressing a button does.

Another key point is that **JMenuItem** is a superclass of **JMenu**. This allows the creation of submenus, which are, essentially, menus within menus. To create a submenu, you first create and populate a **JMenu** object and then add it to another **JMenu** object. You will see this process in action in the following section.

As mentioned in passing previously, when a menu item is selected, an action event is generated. The action command string associated with that action event will, by default, be the name of the selection. Thus, you can determine which item was selected by examining

the action command. Of course, you can also use separate anonymous inner classes or lambda expressions to handle each menu item's action events. In this case, the menu selection is already known, and there is no need to examine the action command string to determine which item was selected.

Menus can also generate other types of events. For example, each time that a menu is activated, selected, or canceled, a **MenuEvent** is generated that can be listened for via a **MenuListener**. Other menu-related events include **MenuKeyEvent**, **MenuDragMouseEvent**, and **PopupMenuEvent**. In many cases, however, you need only watch for action events, and in this chapter, we will use only action events.

An Overview of JMenuBar, JMenu, and JMenuItem

Before you can create a menu, you need to know something about the three core menu classes: **JMenuBar**, **JMenu**, and **JMenuItem**. These form the minimum set of classes needed to construct a main menu for an application. **JMenu** and **JMenuItem** are also used by popup menus. Thus, these classes form the foundation of the menu system.

JMenuBar

As mentioned, **JMenuBar** is essentially a container for menus. Like all components, it inherits **JComponent** (which inherits **Container** and **Component**). It has only one constructor, which is the default constructor. Therefore, initially the menu bar will be empty, and you will need to populate it with menus prior to use. Each application has one and only one menu bar.

JMenuBar defines several methods, but often you will only need to use one: **add()**. The **add()** method adds a **JMenu** to the menu bar. It is shown here:

```
JMenu add(JMenu menu)
```

Here, *menu* is a **JMenu** instance that is added to the menu bar. A reference to the menu is returned. Menus are positioned in the bar from left to right, in the order in which they are added. If you want to add a menu at a specific location, then use this version of **add()**, which is inherited from **Container**:

```
Component add(Component menu, int idx)
```

Here, *menu* is added at the index specified by *idx*. Indexing begins at 0, with 0 being the left-most menu.

In some cases, you might want to remove a menu that is no longer needed. You can do this by calling **remove()**, which is inherited from **Container**. It has these two forms:

```
void remove(Component menu)
```

```
void remove(int idx)
```

Here, *menu* is a reference to the menu to remove, and *idx* is the index of the menu to remove. Indexing begins at zero.

Another method that is sometimes useful is **getMenuCount()**, shown here:

```
int getMenuCount()
```

It returns the number of elements contained within the menu bar.

JMenuBar defines some other methods that you might find helpful in specialized applications. For example, you can obtain an array of references to the menus in the bar by calling **getSubElements()**. You can determine if a menu is selected by calling **isSelected()**.

Once a menu bar has been created and populated, it is added to a **JFrame** by calling **setJMenuBar()** on the **JFrame** instance. (Menu bars *are not* added to the content pane.) The **setJMenuBar()** method is shown here:

```
void setJMenuBar(JMenuBar mb)
```

Here, *mb* is a reference to the menu bar. The menu bar will be displayed in a position determined by the look and feel. Usually, this is at the top of the window.

JMenu

JMenu encapsulates a menu, which is populated with **JMenuItem**s. As mentioned, it is derived from **JMenuItem**. This means that one **JMenu** can be a selection in another **JMenu**. This enables one menu to be a submenu of another. **JMenu** defines a number of constructors. For example, here is the one used in the examples in this chapter:

```
JMenu(String name)
```

This constructor creates a menu that has the title specified by *name*. Of course, you don't have to give a menu a name. To create an unnamed menu, you can use the default constructor:

```
JMenu()
```

Other constructors are also supported. In each case, the menu is empty until menu items are added to it.

JMenu defines many methods. Here is a brief description of some commonly used ones. To add an item to the menu, use the **add()** method, which has a number of forms, including the two shown here:

```
JMenuItem add(JMenuItem item)
```

```
JMenuItem add(Component item, int idx)
```

Here, *item* is the menu item to add. The first form adds the item to the end of the menu. The second form adds the item at the index specified by *idx*. As expected, indexing starts at zero. Both forms return a reference to the item added. As a point of interest, you can also use **insert()** to add menu items to a menu.

You can add a separator (an object of type **JSeparator**) to a menu by calling **addSeparator()**, shown here:

```
void addSeparator()
```

The separator is added onto the end of the menu. You can insert a separator into a menu by calling **insertSeparator()**, shown next:

```
void insertSeparator(int idx)
```

Here, *idx* specifies the zero-based index at which the separator will be added.

You can remove an item from a menu by calling `remove()`. Two of its forms are shown here:

```
void remove(JMenuItem menu)
void remove(int idx)
```

In this case, *menu* is a reference to the item to remove and *idx* is the index of the item to remove.

You can obtain the number of items in the menu by calling `getMenuComponentCount()`, shown here:

```
int getMenuComponentCount()
```

You can get an array of the items in the menu by calling `getMenuComponents()`, shown next:

```
Component[ ] getMenuComponents()
```

An array containing the components is returned.

JMenuItem

JMenuItem encapsulates an element in a menu. This element can be a selection linked to some program action, such as Save or Close, or it can cause a submenu to be displayed. As mentioned, **JMenuItem** is derived from **AbstractButton**, and every item in a menu can be thought of as a special kind of button. Therefore, when a menu item is selected, an action event is generated. (This is similar to the way a **JButton** fires an action event when it is pressed.) **JMenuItem** defines many constructors. The ones used in this chapter are shown here:

```
JMenuItem(String name)
JMenuItem(Icon image)
JMenuItem(String name, Icon image)
JMenuItem(String name, int mnem)
JMenuItem(Action action)
```

The first constructor creates a menu item with the name specified by *name*. The second creates a menu item that displays the image specified by *image*. The third creates a menu item with the name specified by *name* and the image specified by *image*. The fourth creates a menu item with the name specified by *name* and uses the keyboard mnemonic specified by *mnem*. This mnemonic enables you to select an item from the menu by pressing the specified key. The last constructor creates a menu item using the information specified in *action*. A default constructor is also supported.

Because menu items inherit **AbstractButton**, you have access to the functionality provided by **AbstractButton**. One such method that is often useful with menus is `setEnabled()`, which you can use to enable or disable a menu item. It is shown here:

```
void setEnabled(boolean enable)
```

If *enable* is **true**, the menu item is enabled. If *enable* is **false**, the item is disabled and cannot be selected.

Create a Main Menu

Traditionally, the most commonly used menu is the *main menu*. This is the menu defined by the menu bar, and it is the menu that defines all (or nearly all) of the functionality of an application. Fortunately, Swing makes creating and managing the main menu easy. This section shows you how to construct a basic main menu. Subsequent sections will show you how to add options to it.

Constructing the main menu requires several steps. First, create the **JMenuBar** object that will hold the menus. Next, construct each menu that will be in the menu bar. In general, a menu is constructed by first creating a **JMenu** object and then adding **JMenuItem**s to it. After the menus have been created, add them to the menu bar. The menu bar, itself, must then be added to the frame by calling **setJMenuBar()**. Finally, for each menu item, you must add an action listener that handles the action event fired when the menu item is selected.

A good way to understand the process of creating and managing menus is to work through an example. Here is a program that creates a simple menu bar that contains three menus. The first is a standard File menu that contains Open, Close, Save, and Exit selections. The second menu is called Options, and it contains two submenus called Colors and Priority. The third menu is called Help, and it has one item: About. When a menu item is selected, the name of the selection is displayed in a label in the content pane. Sample output is shown in Figure 33-1.

```
// Demonstrate a simple main menu.

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

class MenuDemo implements ActionListener {

    JLabel jlab;

    MenuDemo() {
        // Create a new JFrame container.
        JFrame jfrm = new JFrame("Menu Demo");

```

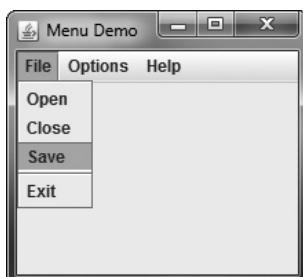


Figure 33-1 Sample output from the **MenuDemo** program

```
// Specify FlowLayout for the layout manager.  
jfrm.setLayout(new FlowLayout());  
  
// Give the frame an initial size.  
jfrm.setSize(220, 200);  
  
// Terminate the program when the user closes the application.  
jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
  
// Create a label that will display the menu selection.  
jlab = new JLabel();  
  
// Create the menu bar.  
JMenuBar jmb = new JMenuBar();  
  
// Create the File menu.  
JMenu jmFile = new JMenu("File");  
JMenuItem jmiOpen = new JMenuItem("Open");  
JMenuItem jmiClose = new JMenuItem("Close");  
JMenuItem jmiSave = new JMenuItem("Save");  
JMenuItem jmiExit = new JMenuItem("Exit");  
jmFile.add(jmiOpen);  
jmFile.add(jmiClose);  
jmFile.add(jmiSave);  
jmFile.addSeparator();  
jmFile.add(jmiExit);  
jmb.add(jmFile);  
  
// Create the Options menu.  
JMenu jmOptions = new JMenu("Options");  
  
// Create the Colors submenu.  
JMenu jmColors = new JMenu("Colors");  
JMenuItem jmiRed = new JMenuItem("Red");  
JMenuItem jmiGreen = new JMenuItem("Green");  
JMenuItem jmiBlue = new JMenuItem("Blue");  
jmColors.add(jmiRed);  
jmColors.add(jmiGreen);  
jmColors.add(jmiBlue);  
jmOptions.add(jmColors);  
  
// Create the Priority submenu.  
JMenu jmPriority = new JMenu("Priority");  
JMenuItem jmiHigh = new JMenuItem("High");  
JMenuItem jmiLow = new JMenuItem("Low");  
jmPriority.add(jmiHigh);  
jmPriority.add(jmiLow);  
jmOptions.add(jmPriority);  
  
// Create the Reset menu item.  
JMenuItem jmiReset = new JMenuItem("Reset");  
jmOptions.addSeparator();  
jmOptions.add(jmiReset);
```

```
// Finally, add the entire options menu to
// the menu bar
jmb.add(jmOptions);

// Create the Help menu.
JMenu jmHelp = new JMenu("Help");
JMenuItem jmiAbout = new JMenuItem("About");
jmHelp.add(jmiAbout);
jmb.add(jmHelp);

// Add action listeners for the menu items.
jmiOpen.addActionListener(this);
jmiClose.addActionListener(this);
jmiSave.addActionListener(this);
jmiExit.addActionListener(this);
jmiRed.addActionListener(this);
jmiGreen.addActionListener(this);
jmiBlue.addActionListener(this);
jmiHigh.addActionListener(this);
jmiLow.addActionListener(this);
jmiReset.addActionListener(this);
jmiAbout.addActionListener(this);

// Add the label to the content pane.
jfrm.add(jlab);

// Add the menu bar to the frame.
jfrm.setJMenuBar(jmb);

// Display the frame.
jfrm.setVisible(true);
}

// Handle menu item action events.
public void actionPerformed(ActionEvent ae) {
    // Get the action command from the menu selection.
    String comStr = ae.getActionCommand();

    // If user chooses Exit, then exit the program.
    if(comStr.equals("Exit")) System.exit(0);

    // Otherwise, display the selection.
    jlab.setText(comStr + " Selected");
}

public static void main(String args[]) {
    // Create the frame on the event dispatching thread.
    SwingUtilities.invokeLater(new Runnable() {
        public void run() {
            new MenuDemo();
        }
    });
}
```

Let's examine, in detail, how the menus in this program are created, beginning with the **MenuDemo** constructor. It starts by creating a **JFrame** and setting its layout manager, size, and default close operation. (These operations are described in Chapter 31.) A **JLabel** is then constructed. It will be used to display a menu selection. Next, the menu bar is constructed and a reference to it is assigned to **jmb** by this statement:

```
// Create the menu bar.  
JMenuBar jmb = new JMenuBar();
```

Then, the File menu **jmFile** and its menu entries are created by this sequence:

```
// Create the File menu.  
JMenu jmFile = new JMenu("File");  
JMenuItem jmiOpen = new JMenuItem("Open");  
JMenuItem jmiClose = new JMenuItem("Close");  
JMenuItem jmiSave = new JMenuItem("Save");  
JMenuItem jmiExit = new JMenuItem("Exit");
```

The names Open, Close, Save, and Exit will be shown as selections in the menu. Next, the menu entries are added to the file menu by this sequence:

```
jmFile.add(jmiOpen);  
jmFile.add(jmiClose);  
jmFile.add(jmiSave);  
jmFile.addSeparator();  
jmFile.add(jmiExit);
```

Finally, the File menu is added to the menu bar with this line:

```
jmb.add(jmFile);
```

Once the preceding code sequence completes, the menu bar will contain one entry: File. The File menu will contain four selections in this order: Open, Close, Save, and Exit. However, notice that a separator has been added before Exit. This visually separates Exit from the preceding three selections.

The Options menu is constructed using the same basic process as the File menu. However, the Options menu consists of two submenus, Colors and Priority, and a Reset entry. The submenus are first constructed individually and then added to the Options menu. The Reset item is added last. Then, the Options menu is added to the menu bar. The Help menu is constructed using the same process.

Notice that **MenuDemo** implements the **ActionListener** interface and action events generated by a menu selection are handled by the **actionPerformed()** method defined by **MenuDemo**. Therefore, the program adds **this** as the action listener for the menu items. Notice that no listeners are added to the Colors or Priority items because they are not actually selections. They simply activate submenus.

Finally, the menu bar is added to the frame by the following line:

```
jfrm.setJMenuBar(jmb);
```

As mentioned, menu bars are not added to the content pane. They are added directly to the **JFrame**.

The **actionPerformed()** method handles the action events generated by the menu. It obtains the action command string associated with the selection by calling **getActionCommand()** on the event. It stores a reference to this string in **comStr**. Then, it tests the action command against "Exit", as shown here:

```
if(comStr.equals("Exit")) System.exit(0);
```

If the action command is "Exit", then the program terminates by calling **System.exit()**. This method causes the immediate termination of a program and passes its argument as a status code to the calling process, which is usually the operating system or the browser. By convention, a status code of zero means normal termination. Anything else indicates that the program terminated abnormally. For all other menu selections, the choice is displayed.

At this point, you might want to experiment a bit with the **MenuDemo** program. Try adding another menu or adding additional items to an existing menu. It is important that you understand the basic menu concepts before moving on because this program will evolve throughout the course of this chapter.

Add Mnemonics and Accelerators to Menu Items

The menu created in the preceding example is functional, but it is possible to make it better. In real applications, a menu usually includes support for keyboard shortcuts because they give an experienced user the ability to select menu items rapidly. Keyboard shortcuts come in two forms: mnemonics and accelerators. As it applies to menus, a *mnemonic* defines a key that lets you select an item from an active menu by typing the key. Thus, a mnemonic allows you to use the keyboard to select an item from a menu that is already being displayed. An *accelerator* is a key that lets you select a menu item without having to first activate the menu.

A mnemonic can be specified for both **JMenuItem** and **JMenu** objects. There are two ways to set the mnemonic for **JMenuItem**. First, it can be specified when an object is constructed using this constructor:

```
JMenuItem(String name, int mnem)
```

In this case, the name is passed in *name* and the mnemonic is passed in *mnem*. Second, you can set the mnemonic by calling **setMnemonic()**. To specify a mnemonic for **JMenu**, you must call **setMnemonic()**. This method is inherited by both classes from **AbstractButton** and is shown next:

```
void setMnemonic(int mnem)
```

Here, *mnem* specifies the mnemonic. It should be one of the constants defined in **java.awt.event.KeyEvent**, such as **KeyEvent.VK_F** or **KeyEvent.VK_Z**. (There is another version of **setMnemonic()** that takes a **char** argument, but it is considered obsolete.) Mnemonics are not case sensitive, so in the case of **VK_A**, typing either *a* or *A* will work.

By default, the first matching letter in the menu item will be underscored. In cases in which you want to underscore a letter other than the first match, specify the index of the letter as an argument to **setDisplayMnemonicIndex()**, which is inherited by both **JMenu** and **JMenuItem** from **AbstractButton**. It is shown here:

```
void setDisplayedMnemonicIndex(int idx)
```

The index of the letter to underscore is specified by *idx*.

An accelerator can be associated with a **JMenuItem** object. It is specified by calling **setAccelerator()**, shown next:

```
void setAccelerator(KeyStroke ks)
```

Here, *ks* is the key combination that is pressed to select the menu item. **KeyStroke** is a class that contains several factory methods that construct various types of keystroke accelerators. The following are three examples:

```
static KeyStroke getKeyStroke(char ch)
static KeyStroke getKeyStroke(Character ch, int modifier)
static KeyStroke getKeyStroke(int ch, int modifier)
```

Here, *ch* specifies the accelerator character. In the first version, the character is specified as a **char** value. In the second, it is specified as an object of type **Character**. In the third, it is a value of type **KeyEvent**, previously described. The value of *modifier* must be one or more of the following constants, defined in the **java.awt.event.InputEvent** class:

InputEvent.ALT_DOWN_MASK	InputEvent.ALT_GRAPH_DOWN_MASK
InputEvent.CTRL_DOWN_MASK	InputEvent.META_DOWN_MASK
InputEvent.SHIFT_DOWN_MASK	

Therefore, if you pass **VK_A** for the key character and **InputEvent.CTRL_DOWN_MASK** for the modifier, the accelerator key combination is **CTRL-A**.

The following sequence adds both mnemonics and accelerators to the File menu created by the **MenuDemo** program in the previous section. After making this change, you can select the File menu by typing **ALT-F**. Then, you can use the mnemonics **O**, **C**, **S**, or **E** to select an option. Alternatively, you can directly select a File menu option by pressing **CTRL-O**, **CTRL-C**, **CTRL-S**, or **CTRL-E**. Figure 33-2 shows how this menu looks when activated.

```
// Create the File menu with mnemonics and accelerators.
JMenu jmFile = new JMenu("File");
jmFile.setMnemonic(KeyEvent.VK_F);

JMenuItem jmiOpen = new JMenuItem("Open",
                                KeyEvent.VK_O);
jmiOpen.setAccelerator(
```

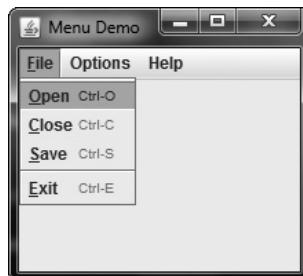


Figure 33-2 The File menu after adding mnemonics and accelerators

```

KeyStroke.getKeyStroke(KeyEvent.VK_O,
                      InputEvent.CTRL_DOWN_MASK) ;

JMenuItem jmiClose = new JMenuItem("Close",
                                  KeyEvent.VK_C) ;
jmiClose.setAccelerator(
    KeyStroke.getKeyStroke(KeyEvent.VK_C,
                          InputEvent.CTRL_DOWN_MASK)) ;

JMenuItem jmiSave = new JMenuItem("Save",
                                 KeyEvent.VK_S) ;
jmiSave.setAccelerator(
    KeyStroke.getKeyStroke(KeyEvent.VK_S,
                          InputEvent.CTRL_DOWN_MASK)) ;

JMenuItem jmiExit = new JMenuItem("Exit",
                                 KeyEvent.VK_E) ;
jmiExit.setAccelerator(
    KeyStroke.getKeyStroke(KeyEvent.VK_E,
                          InputEvent.CTRL_DOWN_MASK)) ;

```

Add Images and Tooltips to Menu Items

You can add images to menu items or use images instead of text. The easiest way to add an image is to specify it when the menu item is being constructed using one of these constructors:

`JMenuItem(Icon image)`

`JMenuItem(String name, Icon image)`

The first creates a menu item that displays the image specified by *image*. The second creates a menu item with the name specified by *name* and the image specified by *image*. For example, here the About menu item is associated with an image when it is created.

```

ImageIcon icon = new ImageIcon("AboutIcon.gif");
JMenuItem jmiAbout = new JMenuItem("About", icon);

```

After this addition, the icon specified by **icon** will be displayed next to the text "About" when the Help menu is displayed. This is shown in Figure 33-3. You can also add an icon to a menu item after the item has been created by calling **setIcon()**, which is inherited from

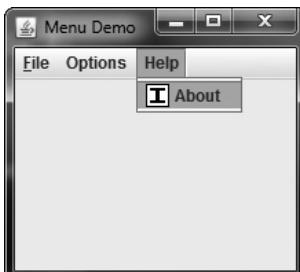


Figure 33-3 The About item with the addition of an icon

AbstractButton. You can specify the horizontal alignment of the image relative to the text by calling `setHorizontalTextPosition()`.

You can specify a disabled icon, which is shown when the menu item is disabled, by calling `setDisabledIcon()`. Normally, when a menu item is disabled, the default icon is shown in gray. If a disabled icon is specified, then that icon is displayed when the menu item is disabled.

A *tooltip* is a small message that describes an item. It is automatically displayed if the mouse remains over the item for a moment. You can add a tooltip to a menu item by calling `setToolTipText()` on the item, specifying the text you want displayed. It is shown here:

```
void setToolTipText(String msg)
```

In this case, *msg* is the string that will be displayed when the tooltip is activated. For example, this creates a tooltip for the About item:

```
jmiAbout.setToolTipText("Info about the MenuDemo program.");
```

As a point of interest, `setToolTipText()` is inherited by **JMenuItem** from **JComponent**. This means you can add a tooltip to other types of components, such as a push button. You might want to try this on your own.

Use JRadioButtonMenuItem and JCheckBoxMenuItem

Although the type of menu items used by the preceding examples are, as a general rule, the most commonly used, Swing defines two others: check boxes and radio buttons. These items can streamline a GUI by allowing a menu to provide functionality that would otherwise require additional, stand-alone components. Also, sometimes, including check boxes or radio buttons in a menu simply seems the most natural place for a specific set of features. Whatever your reason, Swing makes it easy to use check boxes and radio buttons in menus, and both are examined here.

To add a check box to a menu, create a **JCheckBoxMenuItem**. It defines several constructors. This is the one used in this chapter:

```
JCheckBoxMenuItem(String name)
```

Here, *name* specifies the name of the item. The initial state of the check box is unchecked. If you want to specify the initial state, you can use this constructor:

```
JCheckBoxMenuItem(String name, boolean state)
```

In this case, if *state* is **true**, the box is initially checked. Otherwise, it is cleared.

JCheckBoxMenuItem also provides constructors that let you specify an icon. Here is one example:

```
JCheckBoxMenuItem(String name, Icon icon)
```

In this case, *name* specifies the name of the item and the image associated with the item is passed in *icon*. The item is initially unchecked. Other constructors are also supported.

Check boxes in menus work like stand-alone check boxes. For example, they generate action events and item events when their state changes. Check boxes are especially useful in menus when you have options that can be selected and you want to display their selected/deselected status.

A radio button can be added to a menu by creating an object of type **JRadioButtonMenuItem**. **JRadioButtonMenuItem** inherits **JMenuItem**. It provides a rich assortment of constructors. The ones used in this chapter are shown here:

```
JRadioButtonMenuItem(String name)
JRadioButtonMenuItem(String name, boolean state)
```

The first constructor creates an unselected radio button menu item that is associated with the name passed in *name*. The second lets you specify the initial state of the button. If *state* is **true**, the button is initially selected. Otherwise, it is deselected. Other constructors let you specify an icon. Here is one example:

```
JRadioButtonMenuItem(String name, Icon icon, boolean state)
```

This creates a radio button menu item that is associated with the name passed in *name* and the image passed in *icon*. If *state* is **true**, the button is initially selected. Otherwise, it is deselected. Several other constructors are supported.

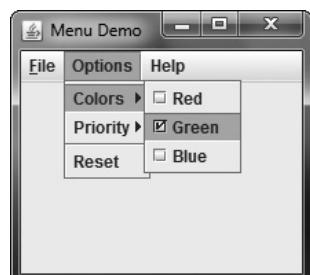
A **JRadioButtonMenuItem** works like a stand-alone radio button, generating item and action events. Like stand-alone radio buttons, menu-based radio buttons must be put into a button group in order for them to exhibit mutually exclusive selection behavior.

Because both **JCheckBoxMenuItem** and **JRadioButtonMenuItem** inherit **JMenuItem**, each has all of the functionality provided by **JMenuItem**. Aside from having the extra capabilities of check boxes and radio buttons, they act like and are used like other menu items.

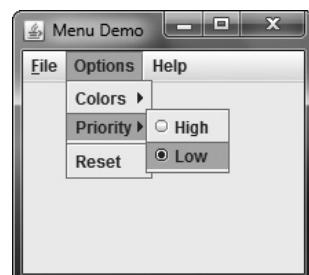
To try check box and radio button menu items, first remove the code that creates the Options menu in the **MenuDemo** example program. Then substitute the following code sequence, which uses check boxes for the Colors submenu and radio buttons for the Priority submenu. After making the substitution, the Options menu will look like those shown in Figure 33-4.

```
// Create the Options menu.
JMenu jmOptions = new JMenu("Options");

// Create the Colors submenu.
JMenu jmColors = new JMenu("Colors");
```



(a)



(b)

Figure 33-4 The effects of check box (a) and radio button (b) menu items

```
// Use check boxes for colors. This allows
// the user to select more than one color.
JCheckBoxMenuItem jmiRed = new JCheckBoxMenuItem("Red");
JCheckBoxMenuItem jmiGreen = new JCheckBoxMenuItem("Green");
JCheckBoxMenuItem jmiBlue = new JCheckBoxMenuItem("Blue");

jmColors.add(jmiRed);
jmColors.add(jmiGreen);
jmColors.add(jmiBlue);
jmOptions.add(jmColors);

// Create the Priority submenu.
JMenu jmPriority = new JMenu("Priority");

// Use radio buttons for the priority setting.
// This lets the menu show which priority is used
// but also ensures that one and only one priority
// can be selected at any one time. Notice that
// the High radio button is initially selected.
JRadioButtonMenuItem jmiHigh =
    new JRadioButtonMenuItem("High", true);
JRadioButtonMenuItem jmiLow =
    new JRadioButtonMenuItem("Low");

jmPriority.add(jmiHigh);
jmPriority.add(jmiLow);
jmOptions.add(jmPriority);

// Create button group for the radio button menu items.
ButtonGroup bg = new ButtonGroup();
bg.add(jmiHigh);
bg.add(jmiLow);

// Create the Reset menu item.
JMenuItem jmiReset = new JMenuItem("Reset");
jmOptions.addSeparator();
jmOptions.add(jmiReset);

// Finally, add the entire options menu to
// the menu bar
jmb.add(jmOptions);
```

Create a Popup Menu

A popular alternative or addition to the menu bar is the popup menu. Typically, a popup menu is activated by clicking the right mouse button when over a component. Popup menus are supported in Swing by the **JPopupMenu** class. **JPopupMenu** has two constructors. In this chapter, only the default constructor is used:

```
JPopupMenu()
```

It creates a default popup menu. The other constructor lets you specify a title for the menu. Whether this title is displayed is subject to the look and feel.

In general, popup menus are constructed like regular menus. First, create a **JPopupMenu** object, and then add menu items to it. Menu item selections are also handled in the same way: by listening for action events. The main difference between a popup menu and regular menu is the activation process.

Activating a popup menu requires three steps.

1. You must register a listener for mouse events.
2. Inside the mouse event handler, you must watch for the popup trigger.
3. When a popup trigger is received, you must show the popup menu by calling **show()**.

Let's examine each of these steps closely.

A popup menu is normally activated by clicking the right mouse button when the mouse pointer is over a component for which a popup menu is defined. Thus, the *popup trigger* is usually caused by right-clicking the mouse on a popup menu–enabled component. To listen for the popup trigger, implement the **MouseListener** interface and then register the listener by calling the **addMouseListener()** method. As described in Chapter 24, **MouseListener** defines the methods shown here:

```
void mouseClicked(MouseEvent me)
void mouseEntered(MouseEvent me)
void mouseExited(MouseEvent me)
void mousePressed(MouseEvent me)
void mouseReleased(MouseEvent me)
```

Of these, two are very important relative to the popup menu: **mousePressed()** and **mouseReleased()**. Depending on the installed look and feel, either of these two events can trigger a popup menu. For this reason, it is often easier to use a **MouseAdapter** to implement the **MouseListener** interface and simply override **mousePressed()** and **mouseReleased()**.

The **MouseEvent** class defines several methods, but only four are commonly needed when activating a popup menu. They are shown here:

```
int getX()
int getY()
boolean isPopupTrigger()
Component getComponent()
```

The current X,Y location of the mouse relative to the source of the event is found by calling **getX()** and **getY()**. These are used to specify the upper-left corner of the popup menu when it is displayed. The **isPopupTrigger()** method returns **true** if the mouse event represents a popup trigger and **false** otherwise. You will use this method to determine when to pop up the menu. To obtain a reference to the component that generated the mouse event, call **getComponent()**.

To actually display the popup menu, call the `show()` method defined by **JPopupMenu**, shown next:

```
void show(Component invoker, int upperX, int upperY)
```

Here, *invoker* is the component relative to which the menu will be displayed. The values of *upperX* and *upperY* define the X,Y location of the upper-left corner of the menu, relative to *invoker*. A common way to obtain the invoker is to call `getComponent()` on the event object passed to the mouse event handler.

The preceding theory can be put into practice by adding a popup Edit menu to the **MenuDemo** program shown at the start of this chapter. This menu will have three items called Cut, Copy, and Paste. Begin by adding the following instance variable to **MenuDemo**:

```
JPopupMenu jpu;
```

The **jpu** variable will hold a reference to the popup menu.

Next, add the following code sequence to the **MenuDemo** constructor:

```
// Create an Edit popup menu.
jpu = new JPopupMenu();

// Create the popup menu items.
JMenuItem jmiCut = new JMenuItem("Cut");
JMenuItem jmiCopy = new JMenuItem("Copy");
JMenuItem jmiPaste = new JMenuItem("Paste");

// Add the menu items to the popup menu.
jpu.add(jmiCut);
jpu.add(jmiCopy);
jpu.add(jmiPaste);

// Add a listener for the popup trigger.
jfrm.addMouseListener(new MouseAdapter() {
    public void mousePressed(MouseEvent me) {
        if(me.isPopupTrigger())
            jpu.show(me.getComponent(), me.getX(), me.getY());
    }
    public void mouseReleased(MouseEvent me) {
        if(me.isPopupTrigger())
            jpu.show(me.getComponent(), me.getX(), me.getY());
    }
});
```

This sequence begins by constructing an instance of **JPopupMenu** and storing it in **jpu**. Then, it creates the three menu items, Cut, Copy, and Paste, in the usual way, and adds them to **jpu**. This finishes the construction of the popup Edit menu. Popup menus are not added to the menu bar or any other object.

Next, a **MouseListener** is added by creating an anonymous inner class. This class is based on the **MouseAdapter** class, which means that the listener need only override those

methods that are relevant to the popup menu: **mousePressed()** and **mouseReleased()**. The adapter provides default implementations of the other **MouseListener** methods. Notice that the mouse listener is added to **jfrm**. This means that a right-button click inside any part of the content pane will trigger the popup menu.

The **mousePressed()** and **mouseReleased()** methods call **isPopupTrigger()** to determine if the mouse event is a popup trigger event. If it is, the popup menu is displayed by calling **show()**. The invoker is obtained by calling **getComponent()** on the mouse event. In this case, the invoker will be the content pane. The X,Y coordinates of the upper-left corner are obtained by calling **getX()** and **getY()**. This makes the menu pop up with its upper-left corner directly under the mouse pointer.

Finally, you also need to add these action listeners to the program. They handle the action events fired when the user selects an item from the popup menu.

```
jmiCut.addActionListener(this);  
jmiCopy.addActionListener(this);  
jmiPaste.addActionListener(this);
```

After you have made these additions, the popup menu can be activated by clicking the right mouse button anywhere inside the content pane of the application. Figure 33-5 shows the result.

One other point about the preceding example. Because the invoker of the popup menu is always **jfrm**, in this case, you could pass it explicitly rather than calling **getComponent()**. To do so, you must make **jfrm** into an instance variable of the **MenuDemo** class (rather than a local variable) so that it is accessible to the inner class. Then you can use this call to **show()** to display the popup menu:

```
jpu.show(jfrm, me.getX(), me.getY());
```

Although this works in this example, the advantage of using **getComponent()** is that the popup menu will automatically pop up relative to the invoking component. Thus, the same code could be used to display any popup menu relative to its invoking object.



Figure 33-5 A popup Edit menu

Create a Toolbar

A toolbar is a component that can serve as both an alternative and as an adjunct to a menu. A toolbar contains a list of buttons (or other components) that give the user immediate access to various program options. For example, a toolbar might contain buttons that select various font options, such as bold, italics, highlight, or underline. These options can be selected without needing to drop through a menu. Typically, toolbar buttons show icons rather than text, although either or both are allowed. Furthermore, tooltips are often associated with icon-based toolbar buttons. Toolbars can be positioned on any side of a window by dragging the toolbar, or they can be dragged out of the window entirely, in which case they become free floating.

In Swing, toolbars are instances of the **JToolBar** class. Its constructors enable you to create a toolbar with or without a title. You can also specify the layout of the toolbar, which will be either horizontal or vertical. The **JToolBar** constructors are shown here:

```
JToolBar( )  
JToolBar(String title)  
JToolBar(int how)  
JToolBar(String title, int how)
```

The first constructor creates a horizontal toolbar with no title. The second creates a horizontal toolbar with the title specified by *title*. The title will show only when the toolbar is dragged out of its window. The third creates a toolbar that is oriented as specified by *how*. The value of *how* must be either **JToolBar.VERTICAL** or **JToolBar.HORIZONTAL**. The fourth constructor creates a toolbar that has the title specified by *title* and is oriented as specified by *how*.

A toolbar is typically used with a window that uses a border layout. There are two reasons for this. First, it allows the toolbar to be initially positioned along one of the four border positions. Frequently, the top position is used. Second, it allows the toolbar to be dragged to any side of the window.

In addition to dragging the toolbar to different locations within a window, you can also drag it out of the window. Doing so creates an *undocked* toolbar. If you specify a title when you create the toolbar, then that title will be shown when the toolbar is undocked.

You add buttons (or other components) to a toolbar in much the same way that you add them to a menu bar. Simply call **add()**. The components are shown in the toolbar in the order in which they are added.

Once you have created a toolbar, you *do not* add it to the menu bar (if one exists). Instead, you add it to the window container. As mentioned, typically you will add a toolbar to the top (that is, north) position of a border layout, using a horizontal orientation. The component that will be affected is added to the center of the border layout. Using this approach causes the program to begin running with the toolbar in the expected location. However, you can drag the toolbar to any of the other positions. Of course, you can also drag the toolbar out of the window.

To illustrate the toolbar, we will add one to the **MenuDemo** program. The toolbar will present three debugging options: set a breakpoint, clear a breakpoint, and resume program execution. Three steps are needed to add the toolbar.

First, remove this line from the program.

```
jfrm.setLayout(new FlowLayout());
```

By removing this line, the **JFrame** automatically uses a border layout.

Second, because **BorderLayout** is being used, change the line that adds the label **jlab** to the frame, as shown next:

```
jfrm.add(jlab, BorderLayout.CENTER);
```

This line explicitly adds **jlab** to the center of the border layout. (Explicitly specifying the center position is technically not necessary because, by default, components are added to the center when a border layout is used. However, explicitly specifying the center makes it clear to anyone reading the code that a border layout is being used and that **jlab** goes in the center.)

Next, add the following code, which creates the Debug toolbar.

```
// Create a Debug toolbar.  
JToolBar jtb = new JToolBar("Debug");  
  
// Load the images.  
ImageIcon set = new ImageIcon("setBP.gif");  
ImageIcon clear = new ImageIcon("clearBP.gif");  
ImageIcon resume = new ImageIcon("resume.gif");  
  
// Create the toolbar buttons.  
 JButton jbtnSet = new JButton(set);  
jbtnSet.setActionCommand("Set Breakpoint");  
jbtnSet.setToolTipText("Set Breakpoint");  
  
 JButton jbtnClear = new JButton(clear);  
jbtnClear.setActionCommand("Clear Breakpoint");  
jbtnClear.setToolTipText("Clear Breakpoint");  
  
 JButton jbtnResume = new JButton(resume);  
jbtnResume.setActionCommand("Resume");  
jbtnResume.setToolTipText("Resume");  
  
// Add the buttons to the toolbar.  
jtb.add(jbtnSet);  
jtb.add(jbtnClear);  
jtb.add(jbtnResume);  
  
// Add the toolbar to the north position of  
// the content pane.  
jfrm.add(jtb, BorderLayout.NORTH);
```

Let's look at this code closely. First, a **JToolBar** is created and given the title "Debug". Then, a set of **ImageIcon** objects are created that hold the images for the toolbar buttons. Next, three toolbar buttons are created. Notice that each has an image, but no text. Also, each is explicitly given an action command and a tooltip. The action commands are set

because the buttons are not given names when they are constructed. Tooltips are especially useful when applied to icon-based toolbar components because sometimes it's hard to design images that are intuitive to all users. The buttons are then added to the toolbar, and the toolbar is added to the north side of the border layout of the frame.

Finally, add the action listeners for the toolbar, as shown here:

```
// Add the toolbar action listeners.  
jbtnSet.addActionListener(this);  
jbtnClear.addActionListener(this);  
jbtnResume.addActionListener(this);
```

Each time the user presses a toolbar button, an action event is fired, and it is handled in the same way as the other menu-related events. Figure 33-6 shows the toolbar in action.

Use Actions

Often, a toolbar and a menu item contain items in common. For example, the same functions provided by the Debug toolbar in the preceding example might also be offered through a menu selection. In such a case, selecting an option (such as setting a breakpoint) causes the same action to occur, independently of whether the menu or the toolbar was used. Also, both the toolbar button and the menu item would (most likely) use the same icon. Furthermore, when a toolbar button is disabled, the corresponding menu item would also need to be disabled. Such a situation would normally lead to a fair amount of duplicated, interdependent code, which is less than optimal. Fortunately, Swing provides a solution: the *action*.

An action is an instance of the **Action** interface. **Action** extends the **ActionListener** interface and provides a means of combining state information with the **actionPerformed()** event handler. This combination allows one action to manage two or more components. For example, an action lets you centralize the control and handling of a toolbar button and a menu item. Instead of having to duplicate code, your program need only create an action that automatically handles both components.

Because **Action** extends **ActionListener**, an action must provide an implementation of the **actionPerformed()** method. This handler will process the action events generated by the objects linked to the action.

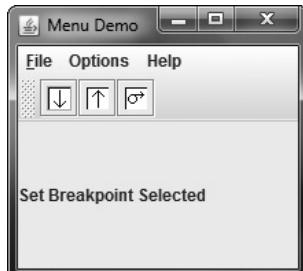


Figure 33-6 The Debug toolbar in action

In addition to the inherited **actionPerformed()** method, **Action** defines several methods of its own. One of particular interest is **putValue()**. It sets the value of the various properties associated with an action and is shown here:

```
void putValue(String key, Object val)
```

It assigns *val* to the property specified by *key* that represents the desired property. Although not used by the example that follows, it is helpful to note that **Action** also supplies the **getValue()** method that obtains a specified property. It is shown here:

```
Object getValue(String key)
```

It returns a reference to the property specified by *key*.

The key values used by **putValue()** and **getValue()** include those shown here:

Key Value	Description
static final String ACCELERATOR_KEY	Represents the accelerator property. Accelerators are specified as KeyStroke objects.
static final String ACTION_COMMAND_KEY	Represents the action command property. An action command is specified as a string.
static final String DISPLAYED_MNEMONIC_INDEX_KEY	Represents the index of the character displayed as the mnemonic. This is an Integer value.
static final String LARGE_ICON_KEY	Represents the large icon associated with the action. The icon is specified as an object of type Icon .
static final String LONG_DESCRIPTION	Represents a long description of the action. This description is specified as a string.
static final String MNEMONIC_KEY	Represents the mnemonic property. A mnemonic is specified as a KeyEvent constant.
static final String NAME	Represents the name of the action (which also becomes the name of the button or menu item to which the action is linked). The name is specified as a string.
static final String SELECTED_KEY	Represents the selection status. If set, the item is selected. The state is represented by a Boolean value.
static final String SHORT_DESCRIPTION	Represents the tooltip text associated with the action. The tooltip text is specified as a string.
static final String SMALL_ICON	Represents the icon associated with the action. The icon is specified as an object of type Icon .

For example, to set the mnemonic to the letter X, use this call to **putValue()**:

```
actionOb.putValue(MNEMONIC_KEY, new Integer(KeyEvent.VK_X));
```

One **Action** property that is not accessible through **putValue()** and **getValue()** is the enabled/disabled status. For this, you use the **setEnabled()** and **isEnabled()** methods. They are shown here:

```
void setEnabled(boolean enabled)  
boolean isEnabled()
```

For **setEnabled()**, if *enabled* is **true**, the action is enabled. Otherwise, it is disabled. If the action is enabled, **isEnabled()** returns **true**. Otherwise, it returns **false**.

Although you can implement all of the **Action** interface yourself, you won't usually need to. Instead, Swing provides a partial implementation called **AbstractAction** that you can extend. By extending **AbstractAction**, you need implement only one method:

actionPerformed(). The other **Action** methods are provided for you. **AbstractAction** provides three constructors. The one used in this chapter is shown here:

```
AbstractAction(String name, Icon image)
```

It constructs an **AbstractAction** that has the name specified by *name* and the icon specified by *image*.

Once you have created an action, it can be added to a **JToolBar** and used to construct a **JMenuItem**. To add an action to a **JToolBar**, use this version of **add()**:

```
void add(Action actObj)
```

Here, *actObj* is the action that is being added to the toolbar. The properties defined by *actObj* are used to create a toolbar button. To create a menu item from an action, use this **JMenuItem** constructor:

```
JMenuItem(Action actObj)
```

Here, *actObj* is the action used to construct a menu item according to its properties.

NOTE In addition to **JToolBar** and **JMenuItem**, actions are also supported by several other Swing components, such as **JPopupMenu**, **JButton**, **JRadioButton**, and **JCheckBox**. **JRadioButtonMenuItem** and **JCheckBoxMenuItem** also support actions.

To illustrate the benefit of actions, we will use them to manage the Debug toolbar created in the previous section. We will also add a Debug submenu under the Options main menu. The Debug submenu will contain the same selections as the Debug toolbar: Set Breakpoint, Clear Breakpoint, and Resume. The same actions that support these items in the toolbar will also support these items in the menu. Therefore, instead of having to create duplicate code to handle both the toolbar and menu, both are handled by the actions.

Begin by creating an inner class called **DebugAction** that extends **AbstractAction**, as shown here:

```
// A class to create an action for the Debug menu
// and toolbar.
class DebugAction extends AbstractAction {
    public DebugAction(String name, Icon image, int mnem,
                       int accel, String tTip) {
        super(name, image);
        putValue(ACCELERATOR_KEY,
                 KeyStroke.getKeyStroke(accel,
                                       InputEvent.CTRL_DOWN_MASK));
        putValue(MNEMONIC_KEY, new Integer(mnem));
        putValue(SHORT_DESCRIPTION, tTip);
    }

    // Handle events for both the toolbar and the
    // Debug menu.
    public void actionPerformed(ActionEvent ae) {
        String comStr = ae.getActionCommand();

        jlab.setText(comStr + " Selected");

        // Toggle the enabled status of the
        // Set and Clear Breakpoint options.
        if(comStr.equals("Set Breakpoint")) {
            clearAct.setEnabled(true);
            setAct.setEnabled(false);
        } else if(comStr.equals("Clear Breakpoint")) {
            clearAct.setEnabled(false);
            setAct.setEnabled(true);
        }
    }
}
```

DebugAction extends **AbstractAction**. It creates an action class that will be used to define the properties associated with the Debug menu and toolbar. Its constructor has five parameters that let you specify the following items:

- Name
- Icon
- Mnemonic
- Accelerator
- Tooltip

The first two are passed to **AbstractAction**'s constructor via **super**. The other three properties are set through calls to **putValue()**.

The **actionPerformed()** method of **DebugAction** handles events for the action. This means that when an instance of **DebugAction** is used to create a toolbar button and a menu

item, events generated by either of those components are handled by the **actionPerformed()** method in **DebugAction**. Notice that this handler displays the selection in **jlab**. In addition, if the Set Breakpoint option is selected, then the Clear Breakpoint option is enabled and the Set Breakpoint option is disabled. If the Clear Breakpoint option is selected, then the Set Breakpoint option is enabled and the Clear Breakpoint option is disabled. This illustrates how an action can be used to enable or disable a component. When an action is disabled, it is disabled for all uses of that action. In this case, if Set Breakpoint is disabled, then it is disabled both in the toolbar and in the menu.

Next, add these **DebugAction** instance variables to **MenuDemo**:

```
DebugAction setAct;
DebugAction clearAct;
DebugAction resumeAct;
```

Next, create three **ImageIcons** that represent the Debug options, as shown here:

```
// Load the images for the actions.
ImageIcon setIcon = new ImageIcon("setBP.gif");
ImageIcon clearIcon = new ImageIcon("clearBP.gif");
ImageIcon resumeIcon = new ImageIcon("resume.gif");
```

Now, create the actions that manage the Debug options, as shown here:

```
// Create actions.
setAct =
    new DebugAction("Set Breakpoint",
                    setIcon,
                    KeyEvent.VK_S,
                    KeyEvent.VK_B,
                    "Set a break point.");

clearAct =
    new DebugAction("Clear Breakpoint",
                    clearIcon,
                    KeyEvent.VK_C,
                    KeyEvent.VK_L,
                    "Clear a break point.");

resumeAct =
    new DebugAction("Resume",
                    resumeIcon,
                    KeyEvent.VK_R,
                    KeyEvent.VK_R,
                    "Resume execution after breakpoint.");

// Initially disable the Clear Breakpoint option.
clearAct.setEnabled(false);
```

Notice that the accelerator for Set Breakpoint is B and the accelerator for Clear Breakpoint is L. The reason these keys are used rather than S and C is that these keys are already allocated by the File menu for Save and Close. However, they can still be used as mnemonics because each mnemonic is localized to its own menu. Also notice that the action that

represents Clear Breakpoint is initially disabled. It will be enabled only after a breakpoint has been set.

Next, use the actions to create buttons for the toolbar and then add those buttons to the toolbar, as shown here:

```
// Create the toolbar buttons by using the actions.  
JButton jbtnSet = new JButton(setAct);  
JButton jbtnClear = new JButton(clearAct);  
JButton jbtnResume = new JButton(resumeAct);  
  
// Create a Debug toolbar.  
JToolBar jtb = new JToolBar("Breakpoints");  
  
// Add the buttons to the toolbar.  
jtb.add(jbtnSet);  
jtb.add(jbtnClear);  
jtb.add(jbtnResume);  
  
// Add the toolbar to the north position of  
// the content pane.  
jfrm.add(jtb, BorderLayout.NORTH);
```

Finally, create the Debug menu, as shown next:

```
// Now, create a Debug menu that goes under the Options  
// menu bar item. Use the actions to create the items.  
JMenu jmDebug = new JMenu("Debug");  
JMenuItem jmiSetBP = new JMenuItem(setAct);  
JMenuItem jmiClearBP = new JMenuItem(clearAct);  
JMenuItem jmiResume = new JMenuItem(resumeAct);  
jmDebug.add(jmiSetBP);  
jmDebug.add(jmiClearBP);  
jmDebug.add(jmiResume);  
jmOptions.add(jmDebug);
```

After making these changes and additions, the actions that you created will be used to manage both the Debug menu and the toolbar. Thus, changing a property in the action (such as disabling it) will affect all uses of that action. The program will now look as shown in Figure 33-7.

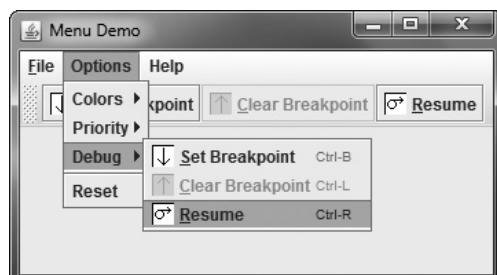


Figure 33-7 Using actions to manage the Debug toolbar and menu

Put the Entire MenuDemo Program Together

Throughout the course of this discussion, many changes and additions have been made to the **MenuDemo** program shown at the start of the chapter. Before concluding, it will be helpful to assemble all the pieces. Doing so not only eliminates any ambiguity about the way the pieces fit together, but it also gives you a complete menu demonstration program that you can experiment with.

The following version of **MenuDemo** includes all of the additions and enhancements described in this chapter. For clarity, the program has been reorganized, with separate methods being used to construct the various menus and toolbar. Notice that several of the menu-related variables, such as **jmb**, **jmFile**, and **jtb**, have been made into instance variables.

```
// The complete MenuDemo program.

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

class MenuDemo implements ActionListener {

    JLabel jlab;
    JMenuBar jmb;
    JToolBar jtb;
    JPopupMenu jpu;

    DebugAction setAct;
    DebugAction clearAct;
    DebugAction resumeAct;

    MenuDemo() {
        // Create a new JFrame container.
        JFrame jfrm = new JFrame("Complete Menu Demo");

        // Use default border layout.

        // Give the frame an initial size.
        jfrm.setSize(360, 200);

        // Terminate the program when the user closes the application.
        jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        // Create a label that will display the menu selection.
        jlab = new JLabel();

        // Create the menu bar.
        jmb = new JMenuBar();

        // Make the File menu.
        makeFileMenu();
    }

    void makeFileMenu() {
        // Create the File menu.
        JMenu jmFile = new JMenu("File");
        jmb.add(jmFile);

        // Add the Set button to the File menu.
        jmFile.add(setAct);
        jmFile.add(clearAct);
        jmFile.add(resumeAct);

        // Add the Exit item to the File menu.
        JMenuItem jmExit = new JMenuItem("Exit");
        jmExit.addActionListener(this);
        jmFile.add(jmExit);
    }

    void actionPerformed(ActionEvent e) {
        if (e.getSource() == setAct) {
            jlab.setText("Set");
        } else if (e.getSource() == clearAct) {
            jlab.setText("Clear");
        } else if (e.getSource() == resumeAct) {
            jlab.setText("Resume");
        } else if (e.getSource() == jmExit) {
            System.exit(0);
        }
    }
}
```

```
// Construct the Debug actions.  
makeActions();  
  
// Make the toolbar.  
makeToolBar();  
  
// Make the Options menu.  
makeOptionsMenu();  
  
// Make the Help menu.  
makeHelpMenu();  
  
// Make the Edit popup menu.  
makeEditPUMenu();  
  
// Add a listener for the popup trigger.  
jfrm.addMouseListener(new MouseAdapter() {  
    public void mousePressed(MouseEvent me) {  
        if(me.isPopupTrigger())  
            jpu.show(me.getComponent(), me.getX(), me.getY());  
    }  
    public void mouseReleased(MouseEvent me) {  
        if(me.isPopupTrigger())  
            jpu.show(me.getComponent(), me.getX(), me.getY());  
    }  
});  
  
// Add the label to the center of the content pane.  
jfrm.add(jlab, SwingConstants.CENTER);  
  
// Add the toolbar to the north position of  
// the content pane.  
jfrm.add(jtb, BorderLayout.NORTH);  
  
// Add the menu bar to the frame.  
jfrm.setJMenuBar(jmb);  
  
// Display the frame.  
jfrm.setVisible(true);  
}  
  
// Handle menu item action events.  
// This does NOT handle events generated  
// by the Debug options.  
public void actionPerformed(ActionEvent ae) {  
    // Get the action command from the menu selection.  
    String comStr = ae.getActionCommand();  
  
    // If user chooses Exit, then exit the program.  
    if(comStr.equals("Exit")) System.exit(0);  
  
    // Otherwise, display the selection.  
    jlab.setText(comStr + " Selected");  
}
```



```
JMenuItem jmiExit = new JMenuItem("Exit",
                                  KeyEvent.VK_E);
jmiExit.setAccelerator(
    KeyStroke.getKeyStroke(KeyEvent.VK_E,
                          InputEvent.CTRL_DOWN_MASK));

jmFile.add(jmiOpen);
jmFile.add(jmiClose);
jmFile.add(jmiSave);
jmFile.addSeparator();
jmFile.add(jmiExit);
jmb.add(jmFile);

// Add the action listeners for the File menu.
jmiOpen.addActionListener(this);
jmiClose.addActionListener(this);
jmiSave.addActionListener(this);
jmiExit.addActionListener(this);
}

// Create the Options menu.
void makeOptionsMenu() {
    JMenu jmOptions = new JMenu("Options");

    // Create the Colors submenu.
    JMenu jmColors = new JMenu("Colors");

    // Use check boxes for colors. This allows
    // the user to select more than one color.
    JCheckBoxMenuItem jmiRed = new JCheckBoxMenuItem("Red");
    JCheckBoxMenuItem jmiGreen = new JCheckBoxMenuItem("Green");
    JCheckBoxMenuItem jmiBlue = new JCheckBoxMenuItem("Blue");

    // Add the items to the Colors menu.
    jmColors.add(jmiRed);
    jmColors.add(jmiGreen);
    jmColors.add(jmiBlue);
    jmOptions.add(jmColors);

    // Create the Priority submenu.
    JMenu jmPriority = new JMenu("Priority");

    // Use radio buttons for the priority setting.
    // This lets the menu show which priority is used
    // but also ensures that one and only one priority
    // can be selected at any one time. Notice that
    // the High radio button is initially selected.
    JRadioButtonMenuItem jmiHigh =
        new JRadioButtonMenuItem("High", true);
    JRadioButtonMenuItem jmiLow =
        new JRadioButtonMenuItem("Low");

    // Add the items to the Priority menu.
    jmPriority.add(jmiHigh);
```

```
jmPriority.add(jmiLow);
jmOptions.add(jmPriority);

// Create a button group for the radio button
// menu items.
ButtonGroup bg = new ButtonGroup();
bg.add(jmiHigh);
bg.add(jmiLow);

// Now, create a Debug submenu that goes under
// the Options menu bar item. Use actions to
// create the items.
JMenu jmDebug = new JMenu("Debug");
JMenuItem jmiSetBP = new JMenuItem(setAct);
JMenuItem jmiClearBP = new JMenuItem(clearAct);
JMenuItem jmiResume = new JMenuItem(resumeAct);

// Add the items to the Debug menu.
jmDebug.add(jmiSetBP);
jmDebug.add(jmiClearBP);
jmDebug.add(jmiResume);
jmOptions.add(jmDebug);

// Create the Reset menu item.
JMenuItem jmiReset = new JMenuItem("Reset");
jmOptions.addSeparator();
jmOptions.add(jmiReset);

// Finally, add the entire options menu to
// the menu bar
jmb.add(jmOptions);

// Add the action listeners for the Options menu,
// except for those supported by the Debug menu.
jmiRed.addActionListener(this);
jmiGreen.addActionListener(this);
jmiBlue.addActionListener(this);
jmiHigh.addActionListener(this);
jmiLow.addActionListener(this);
jmiReset.addActionListener(this);
}

// Create the Help menu.
void makeHelpMenu() {
    JMenu jmHelp = new JMenu("Help");

    // Add an icon to the About menu item.
    ImageIcon icon = new ImageIcon("AboutIcon.gif");

    JMenuItem jmiAbout = new JMenuItem("About", icon);
    jmiAbout.setToolTipText("Info about the MenuDemo program.");
    jmHelp.add(jmiAbout);
    jmb.add(jmHelp);
```

```
// Add action listener for About.  
jmiAbout.addActionListener(this);  
}  
  
// Construct the actions needed by the Debug menu  
// and toolbar.  
void makeActions() {  
    // Load the images for the actions.  
    ImageIcon setIcon = new ImageIcon("setBP.gif");  
    ImageIcon clearIcon = new ImageIcon("clearBP.gif");  
    ImageIcon resumeIcon = new ImageIcon("resume.gif");  
  
    // Create actions.  
    setAct =  
        new DebugAction("Set Breakpoint",  
                        setIcon,  
                        KeyEvent.VK_S,  
                        KeyEvent.VK_B,  
                        "Set a break point.");  
  
    clearAct =  
        new DebugAction("Clear Breakpoint",  
                        clearIcon,  
                        KeyEvent.VK_C,  
                        KeyEvent.VK_L,  
                        "Clear a break point.");  
  
    resumeAct =  
        new DebugAction("Resume",  
                        resumeIcon,  
                        KeyEvent.VK_R,  
                        KeyEvent.VK_R,  
                        "Resume execution after breakpoint.");  
  
    // Initially disable the Clear Breakpoint option.  
    clearAct.setEnabled(false);  
}  
  
// Create the Debug toolbar.  
void makeToolBar() {  
    // Create the toolbar buttons by using the actions.  
    JButton jbtnSet = new JButton(setAct);  
    JButton jbtnClear = new JButton(clearAct);  
    JButton jbtnResume = new JButton(resumeAct);  
  
    // Create the Debug toolbar.  
    jtb = new JToolBar("Breakpoints");  
  
    // Add the buttons to the toolbar.  
    jtb.add(jbtnSet);  
    jtb.add(jbtnClear);  
    jtb.add(jbtnResume);  
}
```

```
// Create the Edit popup menu.  
void makeEditPUMenu() {  
    jpu = new JPopupMenu();  
  
    // Create the popup menu items  
    JMenuItem jmiCut = new JMenuItem("Cut");  
    JMenuItem jmiCopy = new JMenuItem("Copy");  
    JMenuItem jmiPaste = new JMenuItem("Paste");  
  
    // Add the menu items to the popup menu.  
    jpu.add(jmiCut);  
    jpu.add(jmiCopy);  
    jpu.add(jmiPaste);  
  
    // Add the Edit popup menu action listeners.  
    jmiCut.addActionListener(this);  
    jmiCopy.addActionListener(this);  
    jmiPaste.addActionListener(this);  
}  
  
public static void main(String args[]) {  
    // Create the frame on the event dispatching thread.  
    SwingUtilities.invokeLater(new Runnable() {  
        public void run() {  
            new MenuDemo();  
        }  
    });  
}
```

Continuing Your Exploration of Swing

Swing defines a very large GUI toolkit. It has many more features that you will want to explore on your own. For example, it supplies dialog classes, such as **JOptionPane** and **JDialog**, that you can use to streamline the construction of dialog windows. It also provides additional controls beyond those introduced in Chapter 31. Two you will want to explore are **JSpinner** (which creates a spin control) and **JFormattedTextField** (which supports formatted text). You will also want to experiment with defining your own models for the various components. Frankly, the best way to become familiar with Swing's capabilities is to experiment with it.

This page has been intentionally left blank

PART

IV

Introducing GUI Programming with JavaFX

CHAPTER 34

Introducing JavaFX GUI
Programming

CHAPTER 35

Exploring JavaFX Controls

CHAPTER 36

Introducing JavaFX Menus

This page has been intentionally left blank

CHAPTER

34

Introducing JavaFX GUI Programming

Like all successful languages, Java continues to evolve and improve. This also applies to its libraries. One of the most important examples of this evolutionary process is found in its GUI frameworks. As explained earlier in the book, the original GUI framework was the AWT. Because of its several limitations, it was soon followed by Swing, which offered a far superior approach to creating GUIs. Swing was so successful that it has remained the primary Java GUI framework for over a decade. (And a decade is a long time in the fast-moving world of programming!) However, Swing was designed when the enterprise application dominated software development. Today, consumer applications, and especially mobile apps, have risen in importance, and such applications often demand a GUI that has “visual sparkle.” Furthermore, no matter the type of application, the trend is toward more exciting visual effects. To better handle these types of GUIs, a new approach was needed, and this lead to the creation of JavaFX. JavaFX is Java’s next-generation client platform and GUI framework.

JavaFX provides a powerful, streamlined, flexible framework that simplifies the creation of modern, visually exciting GUIs. As such, it is a very large system, and, as was the case with Swing discussed in Part III, it is not possible to describe it fully in this book. Instead, the purpose of this and the next two chapters is to introduce several of its key features and techniques. Once you understand the fundamentals, you will find it easy to explore other aspects of JavaFX on your own.

One question that naturally arises relating to JavaFX is this: Is JavaFX designed as a replacement for Swing? The answer is a qualified Yes. However, given the large amount of Swing legacy code and the legions of programmers who know how to program for Swing, Swing will be in use for a very long time. This is especially true for enterprise applications. Nevertheless, JavaFX has clearly been positioned as the platform of the future. It is expected that, over the next few years, JavaFX will supplant Swing for new projects. JavaFX is something that no Java programmer can afford to ignore.

Before continuing, it is important to mention that the development of JavaFX occurred in two main phases. The original JavaFX was based on a scripting language called *JavaFX Script*. However, JavaFX Script has been discontinued. Beginning with the release of JavaFX 2.0, JavaFX has been programmed in Java itself and provides a comprehensive API. JavaFX

also supports FXML, which can be (but is not required to be) used to specify the user interface. JavaFX has been bundled with Java since JDK 7, update 4. The latest version of JavaFX is JavaFX 8, which is bundled with JDK 8. (The version number is 8 to align with the JDK version. Thus, the numbers 3 through 7 were skipped.) Because, at the time of this writing, JavaFX 8 represents the latest version of JavaFX, it is the version of JavaFX discussed here. Furthermore, when the term *JavaFX* is used in this and the following chapters, it refers to JavaFX 8.

NOTE This and the following two chapters assume that you have a basic understanding of event handling as introduced in Chapter 24, and Swing fundamentals, as described by the preceding three chapters.

JavaFX Basic Concepts

In general, the JavaFX framework has all of the good features of Swing. For example, JavaFX is lightweight. It can also support an MVC architecture. Much of what you already know about creating GUIs using Swing is conceptually applicable to JavaFX. That said, there are significant differences between the two.

From a programmer's point of view, the first differences you notice between JavaFX and Swing are the organization of the framework and the relationship of the main components. Simply put, JavaFX offers a more streamlined, easier-to-use, updated approach. JavaFX also greatly simplifies the rendering of objects because it handles repainting automatically. It is no longer necessary for your program to handle this task manually. The preceding is not intended to imply that Swing is poorly designed. It is not. It is just that the art and science of programming has moved forward, and JavaFX has received the benefits of that evolution. Simply put, JavaFX facilitates a more visually dynamic approach to GUIs.

The JavaFX Packages

The JavaFX elements are contained in packages that begin with the `javafx` prefix. At the time of this writing, there are more than 30 JavaFX packages in its API library. Here are four examples: `javafx.application`, `javafx.stage`, `javafx.scene`, and `javafx.scene.layout`. Although we will only use a few of these packages in this chapter, you will want to spend some time browsing their capabilities. JavaFX offers a wide array of functionality.

The Stage and Scene Classes

The central metaphor implemented by JavaFX is the *stage*. As in the case of an actual stage play, a stage contains a *scene*. Thus, loosely speaking, a stage defines a space and a scene defines what goes in that space. Or, put another way, a stage is a container for scenes and a scene is a container for the items that comprise the scene. As a result, all JavaFX applications have at least one stage and one scene. These elements are encapsulated in the JavaFX API by the **Stage** and **Scene** classes. To create a JavaFX application, you will, at minimum, add at least one **Scene** object to a **Stage**. Let's look a bit more closely at these two classes.

Stage is a top-level container. All JavaFX applications automatically have access to one **Stage**, called the *primary stage*. The primary stage is supplied by the run-time system when a JavaFX application is started. Although you can create other stages, for many applications, the primary stage will be the only one required.

As mentioned, **Scene** is a container for the items that comprise the scene. These can consist of controls, such as push buttons and check boxes, text, and graphics. To create a scene, you will add those elements to an instance of **Scene**.

Nodes and Scene Graphs

The individual elements of a scene are called *nodes*. For example, a push button control is a node. However, nodes can also consist of groups of nodes. Furthermore, a node can have a child node. In this case, a node with a child is called a *parent node* or *branch node*. Nodes without children are terminal nodes and are called leaves. The collection of all nodes in a scene creates what is referred to as a *scene graph*, which comprises a *tree*.

There is one special type of node in the scene graph, called the *root node*. This is the top-level node and is the only node in the scene graph that does not have a parent. Thus, with the exception of the root node, all other nodes have parents, and all nodes either directly or indirectly descend from the root node.

The base class for all nodes is **Node**. There are several other classes that are, either directly or indirectly, subclasses of **Node**. These include **Parent**, **Group**, **Region**, and **Control**, to name a few.

Layouts

JavaFX provides several layout panes that manage the process of placing elements in a scene. For example, the **FlowPane** class provides a flow layout and the **GridPane** class supports a row/column grid-based layout. Several other layouts, such as **BorderPane** (which is similar to the AWT's **BorderLayout**), are available. The layout panes are packaged in `javafx.scene.layout`.

The Application Class and the Life-cycle Methods

A JavaFX application must be a subclass of the **Application** class, which is packaged in `javafx.application`. Thus, your application class will extend **Application**. The **Application** class defines three life-cycle methods that your application can override. These are called **init()**, **start()**, and **stop()**, and are shown here, in the order in which they are called:

```
void init()
abstract void start(Stage primaryStage)
void stop()
```

The **init()** method is called when the application begins execution. It is used to perform various initializations. As will be explained, however, it *cannot* be used to create a stage or build a scene. If no initializations are required, this method need not be overridden because an empty, default version is provided.

The **start()** method is called after **init()**. This is where your application begins and it *can* be used to construct and set the scene. Notice that it is passed a reference to a **Stage** object. This is the stage provided by the run-time system and is the primary stage. (You can also create other stages, but you won't need to for simple applications.) Notice that this method is abstract. Thus, it must be overridden by your application.

When your application is terminated, the `stop()` method is called. It is here that you can handle any cleanup or shutdown chores. In cases in which no such actions are needed, an empty, default version is provided.

Launching a JavaFX Application

To start a free-standing JavaFX application, you must call the `launch()` method defined by **Application**. It has two forms. Here is the one used in this chapter:

```
public static void launch(String ... args)
```

Here, `args` is a possibly empty list of strings that typically specify command-line arguments. When called, `launch()` causes the application to be constructed, followed by calls to `init()` and `start()`. The `launch()` method will not return until after the application has terminated. This version of `launch()` starts the subclass of **Application** from which `launch()` is called. The second form of `launch()` lets you specify a class other than the enclosing class to start.

Before moving on, it is necessary to make an important point: JavaFX applications that have been packaged by using the **javafxpackager** tool (or its equivalent in an IDE) do not need to include a call to `launch()`. However, its inclusion often simplifies the test/debug cycle, and it lets the program be used without the creation of a JAR file. Thus, it is included in all of the JavaFX programs in this book.

A JavaFX Application Skeleton

All JavaFX applications share the same basic skeleton. Therefore, before looking at any more JavaFX features, it will be useful to see what that skeleton looks like. In addition to showing the general form of a JavaFX application, the skeleton also illustrates how to launch the application and demonstrates when the life-cycle methods are called. A message noting when each life-cycle method is called is displayed on the console. The complete skeleton is shown here:

```
// A JavaFX application skeleton.

import javafx.application.*;
import javafx.scene.*;
import javafx.stage.*;
import javafx.scene.layout.*;

public class JavaFXSkel extends Application {

    public static void main(String[] args) {
        System.out.println("Launching JavaFX application.");
        // Start the JavaFX application by calling launch().
        launch(args);
    }

    // Override the init() method.
    public void init() {
```

```
        System.out.println("Inside the init() method.");
    }

    // Override the start() method.
    public void start(Stage myStage) {

        System.out.println("Inside the start() method.");

        // Give the stage a title.
        myStage.setTitle("JavaFX Skeleton.");

        // Create a root node. In this case, a flow layout pane
        // is used, but several alternatives exist.
        FlowPane rootNode = new FlowPane();

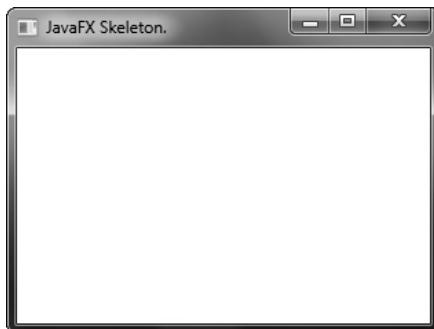
        // Create a scene.
        Scene myScene = new Scene(rootNode, 300, 200);

        // Set the scene on the stage.
        myStage.setScene(myScene);

        // Show the stage and its scene.
        myStage.show();
    }

    // Override the stop() method.
    public void stop() {
        System.out.println("Inside the stop() method.");
    }
}
```

Although the skeleton is quite short, it can be compiled and run. It produces the window shown here:



It also produces the following output on the console:

```
Launching JavaFX application.
Inside the init() method.
Inside the start() method.
```

When you close the window, this message is displayed on the console:

```
Inside the stop() method.
```

Of course, in a real program, the life-cycle methods would not normally output anything to **System.out**. They do so here simply to illustrate when each method is called. Furthermore, as explained earlier, you will need to override the **init()** and **stop()** methods only if your application must perform special startup or shutdown actions. Otherwise, you can use the default implementations of these methods provided by the **Application** class.

Let's examine this program in detail. It begins by importing four packages. The first is **javafx.application**, which contains the **Application** class. The **Scene** class is packaged in **javafx.scene**, and **Stage** is packaged in **javafx.stage**. The **javafx.scene.layout** package provides several layout panes. The one used by the program is **FlowPane**.

Next, the application class **JavaFXSkel** is created. Notice that it extends **Application**. As explained, **Application** is the class from which all JavaFX applications are derived. **JavaFXSkel** contains two methods. The first is **main()**. It is used to launch the application via a call to **launch()**. Notice that the **args** parameter to **main()** is passed to the **launch()** method. Although this is a common approach, you can pass a different set of parameters to **launch()**, or none at all. One other point: As explained earlier, **launch()** is required by a free-standing application, but not in other cases. When it is not needed, **main()** is also not needed. However, for reasons already explained, both **main()** and **launch()** are included in the JavaFX programs in this book.

When the application begins, the **init()** method is called first by the JavaFX run-time system. For the sake of illustration, it simply displays a message on **System.out**, but it would normally be used to initialize some aspect of the application. Of course, if no initialization is required, it is not necessary to override **init()** because an empty, default implementation is provided. It is important to emphasize that **init()** cannot be used to create the stage or scene portions of a GUI. Rather, these items should be constructed and displayed by the **start()** method.

After **init()** finishes, the **start()** method executes. It is here that the initial scene is created and set to the primary stage. Let's look at this method line-by-line. First, notice that **start()** has a parameter of type **Stage**. When **start()** is called, this parameter will receive a reference to the primary stage of the application. It is to this stage that you will set a scene for the application.

After displaying a message on the console that **start()** has begun execution, it sets the title of the stage using this call to **setTitle()**:

```
myStage.setTitle("JavaFX Skeleton.");
```

Although this step is not necessarily required, it is customary for stand-alone applications. This title becomes the name of the main application window.

Next, a root node for a scene is created. The root node is the only node in a scene graph that does not have a parent. In this case, a **FlowPane** is used for the root node, but there are several other classes that can be used for the root.

```
FlowPane rootNode = new FlowPane();
```

As mentioned, a **FlowPane** is a layout manager that uses a flow layout. This is a layout in which elements are positioned line-by-line, with lines wrapping as needed. (Thus, it works much like the **FlowLayout** class used by the AWT and Swing.) In this case, a horizontal flow is used, but it is possible to specify a vertical flow. Although not needed by this skeletal application, it is also possible to specify other layout properties, such as a vertical and horizontal gap between elements, and an alignment. You will see an example of this later in this chapter.

The following line uses the root node to construct a **Scene**:

```
Scene myScene = new Scene(rootNode, 300, 200);
```

Scene provides several versions of its constructor. The one used here creates a scene that has the specified root with the specified width and height. It is shown here:

```
Scene(Parent rootnode, double width, double height)
```

Notice that the type of *rootnode* is **Parent**. It is a subclass of **Node** and encapsulates nodes that can have children. Also notice that the width and the height are **double** values. This lets you pass fractional values, if needed. In the skeleton, the root is **rootNode**, the width is 300 and the height is 200.

The next line in the program sets **myScene** as the scene for **myStage**:

```
myStage.setScene(myScene);
```

Here, **setScene()** is a method defined by **Stage** that sets the scene to that specified by its argument.

In cases in which you don't make further use of the scene, you can combine the previous two steps, as shown here:

```
myStage.setScene(new Scene(rootNode, 300, 200));
```

Because of its compactness, this form will be used by most of the subsequent examples.

The last line in **start()** displays the stage and its scene:

```
myStage.show();
```

In essence, **show()** shows the window that was created by the stage and screen.

When you close the application, its window is removed from the screen and the **stop()** method is called by the JavaFX run-time system. In this case, **stop()** simply displays a message on the console, illustrating when it is called. However, **stop()** would not normally display anything. Furthermore, if your application does not need to handle any shutdown actions, there is no reason to override **stop()** because an empty, default implementation is provided.

Compiling and Running a JavaFX Program

One important advantage of JavaFX is that the same program can be run in a variety of different execution environments. For example, you can run a JavaFX program as a stand-alone desktop application, inside a web browser, or as a Web Start application. However, different ancillary files may be needed in some cases, for example, an HTML file or a Java Network Launch Protocol (JNLP) file.

In general, a JavaFX program is compiled like any other Java program. However, because of the need for additional support for various execution environments, the easiest way to compile a JavaFX application is to use an Integrated Development Environment (IDE) that fully supports JavaFX programming, such as NetBeans. Just follow the instructions for the IDE you are using.

Alternatively, if you want to compile and test a JavaFX application using the command-line tools, you can easily do so. Just compile and run the application in the normal way, using `javac` and `java`. Be aware that using the command-line compiler neither creates any HTML or JNLP files that would be needed if you want to run the application in a way other than as a stand-alone application, nor does it create a JAR file for the program. To create these files, you need to use a tool such as `javafxpackager`.

The Application Thread

In the preceding discussion, it was mentioned that you cannot use the `init()` method to construct a stage or scene. You also cannot create these items inside the application's constructor. The reason is that a stage or scene must be constructed on the *application thread*. However, the application's constructor and the `init()` method are called on the main thread, also called the *launcher thread*. Thus, they can't be used to construct a stage or scene. Instead, you must use the `start()` method, as the skeleton demonstrates, to create the initial GUI because `start()` is called on the application thread.

Furthermore, any changes to the GUI currently displayed must be made from the application thread. Fortunately, in JavaFX, events are sent to your program on the application thread. Therefore, event handlers can be used to interact with the GUI. The `stop()` method is also called on the application thread.

A Simple JavaFX Control: Label

The primary ingredient in most user interfaces is the control because a control enables the user to interact with the application. As you would expect, JavaFX supplies a rich assortment of controls. The simplest control is the label because it just displays a message, which, in this example, is text. Although quite easy to use, the label is a good way to introduce the techniques needed to begin building a scene graph.

The JavaFX label is an instance of the `Label` class, which is packaged in `javafx.scene.control`. `Label` inherits `Labeled` and `Control`, among other classes. The `Labeled` class defines several features that are common to all labeled elements (that is, those that can contain text), and `Control` defines features related to all controls.

`Label` defines three constructors. The one we will use here is

```
Label(String str)
```

Here, `str` is the string that is displayed.

Once you have created a label (or any other control), it must be added to the scene's content, which means adding it to the scene graph. To do this, you will first call `getChildren()`

on the root node of the scene graph. It returns a list of the child nodes in the form of an `ObservableList<Node>`. `ObservableList` is packaged in `javafx.collections`, and it inherits `java.util.List`, which means that it supports all of the features available to a list as defined by the Collections Framework. Using the returned list of child nodes, you can add the label to the list by calling `add()`, passing in a reference to the label.

The following program puts the preceding discussion into action by creating a simple JavaFX application that displays a label:

```
// Demonstrate a JavaFX label.

import javafx.application.*;
import javafx.scene.*;
import javafx.stage.*;
import javafx.scene.layout.*;
import javafx.scene.control.*;

public class JavaFXLabelDemo extends Application {

    public static void main(String[] args) {

        // Start the JavaFX application by calling launch().
        launch(args);
    }

    // Override the start() method.
    public void start(Stage myStage) {

        // Give the stage a title.
        myStage.setTitle("Demonstrate a JavaFX label.");

        // Use a FlowPane for the root node.
        FlowPane rootNode = new FlowPane();

        // Create a scene.
        Scene myScene = new Scene(rootNode, 300, 200);

        // Set the scene on the stage.
        myStage.setScene(myScene);

        // Create a label.
        Label myLabel = new Label("This is a JavaFX label");

        // Add the label to the scene graph.
        rootNode.getChildren().add(myLabel);

        // Show the stage and its scene.
        myStage.show();
    }
}
```

This program produces the following window:



In the program, pay special attention to this line:

```
rootNode.getChildren().add(myLabel);
```

It adds the label to the list of children for which **rootNode** is the parent. Although this line could be separated into its individual pieces if necessary, you will often see it as shown here.

Before moving on, it is useful to point out that **ObservableList** provides a method called **addAll()** that can be used to add two or more children to the scene graph in a single call. (You will see an example of this shortly.) To remove a control from the scene graph, call **remove()** on the **ObservableList**. For example,

```
rootNode.getChildren().remove(myLabel);
```

removes **myLabel** from the scene.

Using Buttons and Events

Although the program in the preceding section presents a simple example of using a JavaFX control and constructing a scene graph, it does not show how to handle events. As you know, most GUI controls generate events that are handled by your program. For example, buttons, check boxes, and lists all generate events when they are used. In many ways, event handling in JavaFX is similar to event handling in Swing or the AWT, but it's more streamlined. Therefore, if you already are proficient at handling events for these other two GUIs, you will have no trouble using the event handling system provided by JavaFX.

One commonly used control is the button. This makes button events one of the most frequently handled. Therefore, a button is a good way to demonstrate the fundamentals of event handling in JavaFX. For this reason, the fundamentals of event handling and the button are introduced together.

Event Basics

The base class for JavaFX events is the **Event** class, which is packaged in **javafx.event**. **Event** inherits **java.util.EventObject**, which means that JavaFX events share the same basic functionality as other Java events. Several subclasses of **Event** are defined. The one that we will use here is **ActionEvent**. It handles action events generated by a button.

In general, JavaFX uses what is, in essence, the delegation event model approach to event handling. To handle an event, you must first register the handler that acts as a listener for the event. When the event occurs, the listener is called. It must then respond to the event and return. In this regard, JavaFX events are managed much like Swing events, for example.

Events are handled by implementing the **EventHandler** interface, which is also in **javafx.event**. It is a generic interface with the following form:

```
interface EventHandler<T extends Event>
```

Here, **T** specifies the type of event that the handler will handle. It defines one method, called **handle()**, which receives the event object as a parameter. It is shown here:

```
void handle(T eventObj)
```

Here, *eventObj* is the event that was generated. Typically, event handlers are implemented through anonymous inner classes or lambda expressions, but you can use stand-alone classes for this purpose if it is more appropriate to your application (for example, if one event handler will handle events from more than one source).

Although not required by the examples in this chapter, it is sometimes useful to know the source of an event. This is especially true if you are using one handler to handle events from different sources. You can obtain the source of the event by calling **getSource()**, which is inherited from **java.util.EventObject**. It is shown here:

```
Object getSource()
```

Other methods in **Event** let you obtain the event type, determine if the event has been consumed, consume an event, fire an event, and obtain the target of the event. When an event is consumed, it stops the event from being passed to a parent handler.

One last point: In JavaFX, events are processed via an *event dispatch chain*. When an event is generated, it is passed to the root node of the chain. The event is then passed down the chain to the target of the event. After the target node processes the event, the event is passed back up the chain, thus allowing parent nodes a chance to process the event, if necessary. This is called *event bubbling*. It is possible for a node in the chain to consume an event, which prevents it from being further processed.

NOTE Although not used in this introduction to JavaFX, an application can also implement an *event filter*, which can be used to manage events. A filter is added to a node by calling **addEventFilter()**, which is defined by **Node**. A filter can consume an event, thus preventing further processing.

Introducing the Button Control

In JavaFX, the push button control is provided by the **Button** class, which is in **javafx.scene.control**. **Button** inherits a fairly long list of base classes that include **ButtonBase**, **Labeled**, **Region**, **Control**, **Parent**, and **Node**. If you examine the API documentation for **Button**, you

will see that much of its functionality comes from its base classes. Furthermore, it supports a wide array of options. However, here we will use its default form. Buttons can contain text, graphics, or both. In this chapter, we will use text-based buttons. An example of a graphics-based button is shown in the next chapter.

Button defines three constructors. The one we will use is shown here:

```
Button(String str)
```

In this case, *str* is the message that is displayed in the button.

When a button is pressed, an **ActionEvent** is generated. **ActionEvent** is packaged in **javafx.event**. You can register a listener for this event by using **setOnAction()**, which has this general form:

```
final void setOnAction(EventHandler<ActionEvent> handler)
```

Here, *handler* is the handler being registered. As mentioned, often you will use an anonymous inner class or lambda expression for the handler. The **setOnAction()** method sets the property **onAction**, which stores a reference to the handler. As with all other Java event handling, your handler must respond to the event as fast as possible and then return. If your handler consumes too much time, it will noticeably slow down the application. For lengthy operations, you must use a separate thread of execution.

Demonstrating Event Handling and the Button

The following program demonstrates event handling. It uses two buttons and a label. Each time a button is pressed, the label is set to display which button was pressed.

```
// Demonstrate JavaFX events and buttons.

import javafx.application.*;
import javafx.scene.*;
import javafx.stage.*;
import javafx.scene.layout.*;
import javafx.scene.control.*;
import javafx.event.*;
import javafx.geometry.*;

public class JavaFXEventDemo extends Application {

    Label response;

    public static void main(String[] args) {

        // Start the JavaFX application by calling launch().
        launch(args);
    }

    // Override the start() method.
    public void start(Stage myStage) {

        // Give the stage a title.
        myStage.setTitle("Demonstrate JavaFX Buttons and Events.");
    }
}
```

```
// Use a FlowPane for the root node. In this case,
// vertical and horizontal gaps of 10.
FlowPane rootNode = new FlowPane(10, 10);

// Center the controls in the scene.
rootNode.setAlignment(Pos.CENTER);

// Create a scene.
Scene myScene = new Scene(rootNode, 300, 100);

// Set the scene on the stage.
myStage.setScene(myScene);

// Create a label.
response = new Label("Push a Button");

// Create two push buttons.
Button btnAlpha = new Button("Alpha");
Button btnBeta = new Button("Beta");

// Handle the action events for the Alpha button.
btnAlpha.setOnAction(new EventHandler<ActionEvent>() {
    public void handle(ActionEvent ae) {
        response.setText("Alpha was pressed.");
    }
});

// Handle the action events for the Beta button.
btnBeta.setOnAction(new EventHandler<ActionEvent>() {
    public void handle(ActionEvent ae) {
        response.setText("Beta was pressed.");
    }
});

// Add the label and buttons to the scene graph.
rootNode.getChildren().addAll(btnAlpha, btnBeta, response);

// Show the stage and its scene.
myStage.show();
}
```

Sample output from this program is shown here:



Let's examine a few key portions of this program. First, notice how buttons are created by these two lines:

```
Button btnAlpha = new Button("Alpha");
Button btnBeta = new Button("Beta");
```

This creates two text-based buttons. The first displays the string Alpha; the second displays Beta.

Next, an action event handler is set for each of these buttons. The sequence for the Alpha button is shown here:

```
// Handle the action events for the Alpha button.
btnAlpha.setOnAction(new EventHandler<ActionEvent>() {
    public void handle(ActionEvent ae) {
        response.setText("Alpha was pressed.");
    }
});
```

As explained, buttons respond to events of type **ActionEvent**. To register a handler for these events, the **setOnAction()** method is called on the button. It uses an anonymous inner class to implement the **EventHandler** interface. (Recall that **EventHandler** defines only the **handle()** method.) Inside **handle()**, the text in the **response** label is set to reflect the fact that the Alpha button was pressed. Notice that this is done by calling the **setText()** method on the label. Events are handled by the Beta button in the same way.

Note that **response** is declared as a field within **FXEventDemo**, rather than as a local variable. This is because it is accessed within the button event handlers, which are anonymous inner classes.

After the event handlers have been set, the **response** label and the buttons **btnAlpha** and **btnBeta** are added to the scene graph by using a call to **addAll()**:

```
rootNode.getChildren().addAll(btnAlpha, btnBeta, response);
```

The **addAll()** method adds a list of nodes to the invoking parent node. Of course, these nodes could have been added by three separate calls to **add()**, but the **addAll()** method is more convenient to use in this situation.

There are two other things of interest in this program that relate to the way the controls are displayed in the window. First, when the root node is created, this statement is used:

```
FlowPane rootNode = new FlowPane(10, 10);
```

Here, the **FlowPane** constructor is passed two values. These specify the horizontal and vertical gap that will be left around elements in the scene. If these gaps are not specified, then two elements (such as two buttons) would be positioned in such a way that no space is between them. Thus, the controls would run together, creating a very unappealing user interface. Specifying gaps prevents this.

The second point of interest is the following line, which sets the alignment of the elements in the **FlowPane**:

```
rootNode.setAlignment(Pos.CENTER);
```

Here, the alignment of the elements is centered. This is done by calling `setAlignment()` on the `FlowPane`. The value `Pos.CENTER` specifies that both a vertical and horizontal center will be used. Other alignments are possible. `Pos` is an enumeration that specifies alignment constants. It is packaged in `javafx.geometry`.

Before moving on, one more point needs to be made. The preceding program used anonymous inner classes to handle button events. However, because the `EventHandler` interface defines only one abstract method, `handle()`, a lambda expression could have passed to `setOnAction()`, instead. In this case, the parameter type of `setOnAction()` would supply the target context for the lambda expression. For example, here is the handler for the Alpha button, rewritten to use a lambda:

```
btnAlpha.setOnAction( (ae) ->
    response.setText("Alpha was pressed.")
);
```

Notice that the lambda expression is more compact than the anonymous inner class. Because lambda expressions are a new feature just recently added to Java, but the anonymous inner class is a widely used construct, readily understood by nearly all Java programmers, the event handlers in subsequent examples will use anonymous inner classes. This will also allow the examples to be compiled by readers using JDK 7 (which does not support lambdas). However, on your own, you might want to experiment with converting them to lambda expressions. It is a good way to gain experience using lambdas in your own code.

Drawing Directly on a Canvas

As mentioned early on, JavaFX handles rendering tasks for you automatically, rather than you handling them manually. This is one of the most important ways that JavaFX improves on Swing. As you may know, in Swing or the AWT, you must call the `repaint()` method to cause a window to be repainted. Furthermore, your application needs to store the window contents, redrawing them when painting is requested. JavaFX eliminates this tedious mechanism because it keeps track of what you display in a scene and redisplays that scene as needed. This is called *retained mode*. With this approach, there is no need to call a method like `repaint()`. Rendering is automatic.

One place that JavaFX's approach to rendering is especially helpful is when displaying graphics objects, such as lines, circles, and rectangles. JavaFX's graphics methods are found in the `GraphicsContext` class, which is part of `java.scene.canvas`. These methods can be used to draw directly on the surface of a canvas, which is encapsulated by the `Canvas` class in `java.scene.canvas`. When you draw something, such as a line, on a canvas, JavaFX automatically renders it whenever it needs to be redisplayed.

Before you can draw on a canvas, you must perform two steps. First, you must create a `Canvas` instance. Second, you must obtain a `GraphicsContext` object that refers to that canvas. You can then use the `GraphicsContext` to draw output on the canvas.

The `Canvas` class is derived from `Node`; thus it can be used as a node in a scene graph. `Canvas` defines two constructors. One is the default constructor, and the other is the one shown here:

```
Canvas(double width, double height)
```

Here, `width` and `height` specify the dimensions of the canvas.

To obtain a **GraphicsContext** that refers to a canvas, call **getGraphicsContext2D()**. Here is its general form:

```
GraphicsContext getGraphicsContext2D( )
```

The graphics context for the canvas is returned.

GraphicsContext defines a large number of methods that draw shapes, text, and images, and support effects and transforms. If sophisticated graphics programming is in your future, you will definitely want to study its capabilities closely. For our purposes, we will use only a few of its methods, but they will give you a sense of its power. They are described here.

You can draw a line using **strokeLine()**, shown here:

```
void strokeLine(double startX, double startY, double endX, double endY)
```

It draws a line from *startX,startY* to *endX,endY*, using the current stroke, which can be a solid color or some more complex style.

To draw a rectangle, use either **strokeRect()** or **fillRect()**, shown here:

```
void strokeRect(double topX, double topY, double width, double height)
```

```
void fillRect(double topX, double topY, double width, double height)
```

The upper-left corner of the rectangle is at *topX,topY*. The *width* and *height* parameters specify its width and height. The **strokeRect()** method draws the outline of a rectangle using the current stroke, and **fillRect()** fills the rectangle with the current fill. The current fill can be as simple as a solid color or something more complex.

To draw an ellipse, use either **strokeOval()** or **fillOval()**, shown next:

```
void strokeOval(double topX, double topY, double width, double height)
```

```
void fillOval(double topX, double topY, double width, double height)
```

The upper-left corner of the rectangle that bounds the ellipse is at *topX,topY*. The *width* and *height* parameters specify its width and height. The **strokeOval()** method draws the outline of an ellipse using the current stroke, and **fillOval()** fills the oval with the current fill. To draw a circle, pass the same value for *width* and *height*.

You can draw text on a canvas by using the **strokeText()** and **fillText()** methods. We will use this version of **fillText()**:

```
void fillText(String str, double topX, double topY)
```

It displays *str* starting at the location specified by *topX,topY*, filling the text with the current fill.

You can set the font and font size of the text being displayed by using **setFont()**. You can obtain the font used by the canvas by calling **getFont()**. By default, the system font is used. You can create a new font by constructing a **Font** object. **Font** is packaged in **javafx.scene.text**. For example, you can create a default font of a specified size by using this constructor:

```
Font(double fontSize)
```

Here, *fontSize* specifies the size of the font.

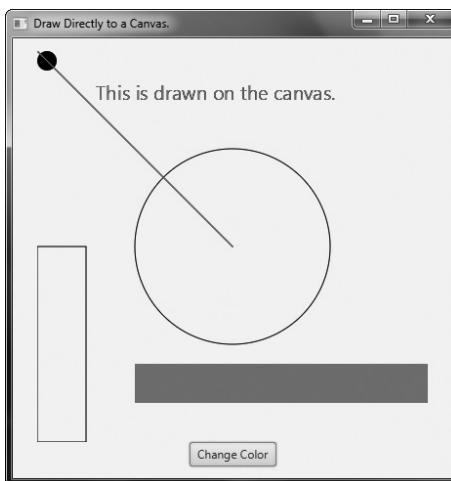
You can specify the fill and stroke using these two methods defined by **Canvas**:

```
void setFill(Paint newFill)
```

```
void setStroke(Paint newStroke)
```

Notice that the parameter of both methods is of type **Paint**. This is an abstract class packaged in **javafx.scene.paint**. Its subclasses define fills and strokes. The one we will use is **Color**, which simply describes a solid color. **Color** defines several static fields that specify a wide array of colors, such as **Color.BLUE**, **Color.RED**, **Color.GREEN**, and so on.

The following program uses the aforementioned methods to demonstrate drawing on a canvas. It first displays a few graphic objects on the canvas. Then, each time the Change Color button is pressed, the color of three of the objects changes color. If you run the program, you will see that the shapes whose color is not changed are unaffected by the change in color of the other objects. Furthermore, if you try covering and then uncovering the window, you will see that the canvas is automatically repainted, without any other actions on the part of your program. Sample output is shown here:



```
// Demonstrate drawing.

import javafx.application.*;
import javafx.scene.*;
import javafx.stage.*;
import javafx.scene.layout.*;
import javafx.scene.control.*;
import javafx.event.*;
import javafx.geometry.*;
import javafx.scene.shape.*;
import javafx.scene.canvas.*;
import javafx.scene.paint.*;
import javafx.scene.text.*;

public class DirectDrawDemo extends Application {

    GraphicsContext gc;
```

```
Color[] colors = { Color.RED, Color.BLUE, Color.GREEN, Color.BLACK };  
int colorIdx = 0;  
  
public static void main(String[] args) {  
  
    // Start the JavaFX application by calling launch().  
    launch(args);  
}  
  
// Override the start() method.  
public void start(Stage myStage) {  
  
    // Give the stage a title.  
    myStage.setTitle("Draw Directly to a Canvas.");  
  
    // Use a FlowPane for the root node.  
    FlowPane rootNode = new FlowPane();  
  
    // Center the nodes in the scene.  
    rootNode.setAlignment(Pos.CENTER);  
  
    // Create a scene.  
    Scene myScene = new Scene(rootNode, 450, 450);  
  
    // Set the scene on the stage.  
    myStage.setScene(myScene);  
  
    // Create a canvas.  
    Canvas myCanvas = new Canvas(400, 400);  
  
    // Get the graphics context for the canvas.  
    gc = myCanvas.getGraphicsContext2D();  
  
    // Create a push button.  
    Button btnChangeColor = new Button("Change Color");  
  
    // Handle the action events for the Change Color button.  
    btnChangeColor.setOnAction(new EventHandler<ActionEvent>() {  
        public void handle(ActionEvent ae) {  
  
            // Set the stroke and fill color.  
            gc.setStroke(colors[colorIdx]);  
            gc.setFill(colors[colorIdx]);  
  
            // Redraw the line, text, and filled rectangle in the  
            // new color. This leaves the color of the other nodes  
            // unchanged.  
            gc.strokeLine(0, 0, 200, 200);  
            gc.fillText("This is drawn on the canvas.", 60, 50);  
            gc.fillRect(100, 320, 300, 40);  
        }  
    });  
}
```

```
// Change the color.  
colorIdx++;  
if(colorIdx == colors.length) colorIdx= 0;  
}  
});  
  
// Draw initial output on the canvas.  
gc.strokeLine(0, 0, 200, 200);  
gc.strokeOval(100, 100, 200, 200);  
gc.strokeRect(0, 200, 50, 200);  
gc.fillOval(0, 0, 20, 20);  
gc.fillRect(100, 320, 300, 40);  
  
// Set the font size to 20 and draw text.  
gc.setFont(new Font(20));  
gc.fillText("This is drawn on the canvas.", 60, 50);  
  
// Add the canvas and button to the scene graph.  
rootNode.getChildren().addAll(myCanvas, btnChangeColor);  
  
// Show the stage and its scene.  
myStage.show();  
}  
}
```

It is important to emphasize that **GraphicsContext** supports many more operations than those demonstrated by the preceding program. For example, you can apply various transforms, rotations, and effects. Despite its power, its various features are easy to master and use. One other point: A canvas is transparent. Therefore, if you stack canvases, the contents of both will show. This may be useful in some situations.

NOTE `javafx.scene.shape` contains several classes that can also be used to draw various types of graphical shapes, such as circles, arcs, and lines. These are represented by nodes and can, therefore, be directly part of the scene graph. You will want to explore these on your own.

This page has been intentionally left blank

CHAPTER

35

Exploring JavaFX Controls

The previous chapter described several of the core concepts relating to the JavaFX GUI framework. In the process, it introduced two controls: the label and the button. This chapter continues the discussion of JavaFX controls. It begins by describing how to include images in a label and button. It then presents an overview of several more JavaFX controls, including check boxes, lists, and trees. Keep in mind that JavaFX is a rich and powerful framework. The purpose of this chapter is to introduce a representative sampling of the JavaFX controls and to describe several common techniques. Once you understand the basics, you will be able to easily learn the other controls.

The JavaFX control classes discussed in this chapter are shown here:

Button	ListView	TextField
CheckBox	RadioButton	ToggleButton
Label	ScrollPane	TreeView

These and the other JavaFX controls are packaged in **javafx.scene.control**.

Also discussed are the **Image** and **ImageView** classes, which provide support for images in controls; **Tooltip**, which is used to add tooltips to a control; as well as various effects and transforms.

Using Image and ImageView

Several of JavaFX's controls let you include an image. For example, in addition to text, you can specify an image in a label or a button. Furthermore, you can embed stand-alone images in a scene directly. At the foundation for JavaFX's support for images are two classes: **Image** and **ImageView**. **Image** encapsulates the image, itself, and **ImageView** manages the display of an image. Both classes are packaged in **javafx.scene.image**.

The **Image** class loads an image from either an **InputStream**, a URL, or a path to the image file. **Image** defines several constructors; this is the one we will use:

```
Image(String url)
```

Here, *url* specifies a URL or a path to a file that supplies the image. The argument is assumed to refer to a path if it does not constitute a properly formed URL. Otherwise, the image is loaded from the URL. The examples that follow will load images from files on the local file system. Other constructors let you specify various options, such as the image's width and height. One other point: **Image** is not derived from **Node**. Thus, it cannot, itself, be part of a scene graph.

Once you have an **Image**, you will use **ImageView** to display it. **ImageView** is derived from **Node**, which means that it can be part of a scene graph. **ImageView** defines three constructors. The first one we will use is shown here:

```
ImageView(Image image)
```

This constructor creates an **ImageView** that uses *image* for its image.

Putting the preceding discussion into action, here is a program that loads an image of an hourglass and displays it via **ImageView**. The hourglass image is contained in a file called **hourglass.png**, which is assumed to be in the local directory.

```
// Load and display an image.

import javafx.application.*;
import javafx.scene.*;
import javafx.stage.*;
import javafx.scene.layout.*;
import javafx.geometry.*;
import javafx.scene.image.*;

public class ImageDemo extends Application {

    public static void main(String[] args) {

        // Start the JavaFX application by calling launch().
        launch(args);
    }

    // Override the start() method.
    public void start(Stage myStage) {

        // Give the stage a title.
        myStage.setTitle("Display an Image");

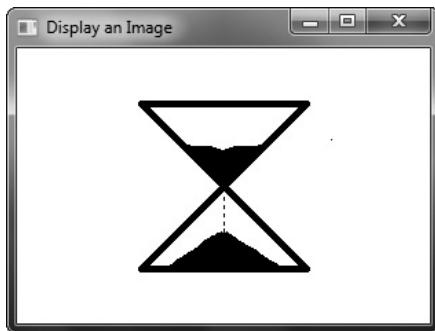
        // Use a FlowPane for the root node.
        FlowPane rootNode = new FlowPane();

        // Use center alignment.
        rootNode.setAlignment(Pos.CENTER);

        // Create a scene.
        Scene myScene = new Scene(rootNode, 300, 200);
```

```
// Set the scene on the stage.  
myStage.setScene(myScene);  
  
// Create an image.  
Image hourglass = new Image("hourglass.png");  
  
// Create an image view that uses the image.  
ImageView hourglassIV = new ImageView(hourglass);  
  
// Add the image to the scene graph.  
rootNode.getChildren().add(hourglassIV);  
  
// Show the stage and its scene.  
myStage.show();  
}  
}
```

Sample output from the program is shown here:



In the program, pay close attention to the following sequence that loads the image and then creates an **ImageView** that uses that image:

```
// Create an image.  
Image hourglass = new Image("HourGlass.png");  
  
// Create an image view that uses the image.  
ImageView hourglassIV = new ImageView(hourglass);
```

As explained, an image by itself cannot be added to the scene graph. It must first be embedded in an **ImageView**.

In cases in which you won't make further use of the image, you can specify a URL or filename when creating an **ImageView**. In this case, there is no need to explicitly create an **Image**. Instead, an **Image** instance containing the specified image is constructed automatically and embedded in the **ImageView**. Here is the **ImageView** constructor that does this:

```
ImageView(String url)
```

Here, *url* specifies the URL or the path to a file that contains the image.

Adding an Image to a Label

As explained in the previous chapter, the **Label** class encapsulates a label. It can display a text message, a graphic, or both. So far, we have used it to display only text, but it is easy to add an image. To do so, use this form of **Label**'s constructor:

```
Label(String str, Node image)
```

Here, *str* specifies the text message and *image* specifies the image. Notice that the image is of type **Node**. This allows great flexibility in the type of image added to the label, but for our purposes, the image type will be **ImageView**.

Here is a program that demonstrates a label that includes a graphic. It creates a label that displays the string "Hourglass" and shows the image of an hourglass that is loaded from the **hourglass.png** file.

```
// Demonstrate an image in a label.

import javafx.application.*;
import javafx.scene.*;
import javafx.stage.*;
import javafx.scene.layout.*;
import javafx.scene.control.*;
import javafx.geometry.*;
import javafx.scene.image.*;

public class LabelImageDemo extends Application {

    public static void main(String[] args) {

        // Start the JavaFX application by calling launch().
        launch(args);
    }

    // Override the start() method.
    public void start(Stage myStage) {

        // Give the stage a title.
        myStage.setTitle("Use an Image in a Label");

        // Use a FlowPane for the root node.
        FlowPane rootNode = new FlowPane();

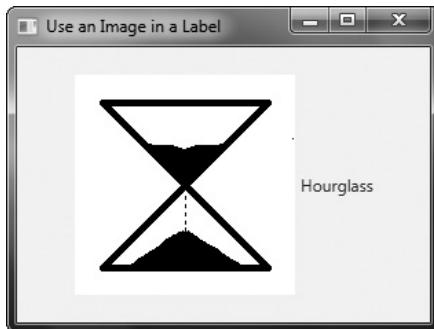
        // Use center alignment.
        rootNode.setAlignment(Pos.CENTER);

        // Create a scene.
        Scene myScene = new Scene(rootNode, 300, 200);

        // Set the scene on the stage.
        myStage.setScene(myScene);
    }
}
```

```
// Create an ImageView that contains the specified image.  
ImageView hourglassIV = new ImageView("hourglass.png");  
  
// Create a label that contains both an image and text.  
Label hourglassLabel = new Label("Hourglass", hourglassIV);  
  
// Add the label to the scene graph.  
rootNode.getChildren().add(hourglassLabel);  
  
// Show the stage and its scene.  
myStage.show();  
}  
}
```

Here is the window produced by the program:



As you can see, both the image and the text are displayed. Notice that the text is to the right of the image. This is the default. You can change the relative positions of the image and text by calling `setContentDisplay()` on the label. It is shown here:

```
final void setContentDisplay(ContentDisplay position)
```

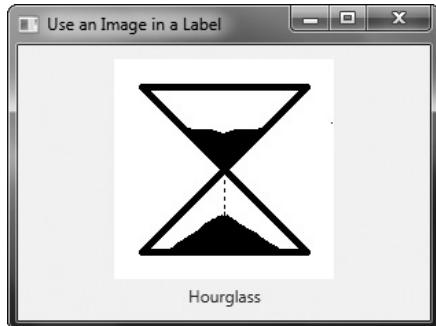
The value passed to `position` determines how the text and image is displayed. It must be one of these values, which are defined by the `ContentDisplay` enumeration:

BOTTOM	RIGHT
CENTER	TEXT_ONLY
GRAPHIC_ONLY	TOP
LEFT	

With the exception of `TEXT_ONLY` and `GRAPHIC_ONLY`, the values specify the location of the image. For example, if you add this line to the preceding program:

```
hourglassLabel.setContentDisplay(ContentDisplay.TOP);
```

the image of the hourglass will be above the text, as shown here:



The other two values let you display either just the text or just the image. This might be useful if your application wants to use an image at some times, and not at others, for example. (If you want only an image, you can simply display it without using a label, as described in the previous section.)

You can also add an image to a label after it has been constructed by using the **setGraphic()** method. It is shown here:

```
final void setGraphic(Node image)
```

Here, *image* specifies the image to add.

Using an Image with a Button

Button is JavaFX's class for push buttons. The preceding chapter introduced the **Button** class. There, you saw an example of a button that contained text. Although such buttons are common, you are not limited to this approach because you can include an image. You can also use only the image if you choose. The procedure for adding an image to a button is similar to that used to add an image to a label. First obtain an **ImageView** of the image. Then add it to the button. One way to add the image is to use this constructor:

```
Button(String str, Node image)
```

Here, *str* specifies the text that is displayed within the button and *image* specifies the image. You can specify the position of the image relative to the text by using **setContentDisplay()** in the same way as just described for **Label**.

Here is an example that displays two buttons that contain images. The first shows an hourglass. The second shows an analog clock. When a button is pressed, the selected timepiece is reported. Notice that the text is displayed beneath the image.

```
// Use an image with a button.
```

```
import javafx.application.*;
import javafx.scene.*;
import javafx.stage.*;
import javafx.scene.layout.*;
```

```
import javafx.scene.control.*;
import javafx.event.*;
import javafx.geometry.*;
import javafx.scene.image.*;

public class ButtonImageDemo extends Application {

    Label response;

    public static void main(String[] args) {

        // Start the JavaFX application by calling launch().
        launch(args);
    }

    // Override the start() method.
    public void start(Stage myStage) {

        // Give the stage a title.
        myStage.setTitle("Use Images with Buttons");

        // Use a FlowPane for the root node. In this case,
        // vertical and horizontal gaps of 10.
        FlowPane rootNode = new FlowPane(10, 10);

        // Center the controls in the scene.
        rootNode.setAlignment(Pos.CENTER);

        // Create a scene.
        Scene myScene = new Scene(rootNode, 250, 450);

        // Set the scene on the stage.
        myStage.setScene(myScene);

        // Create a label.
        response = new Label("Push a Button");

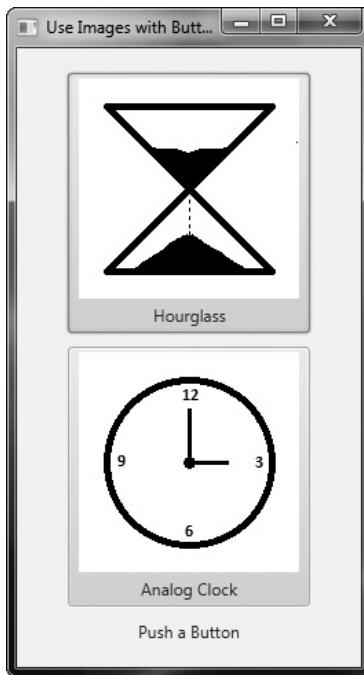
        // Create two image-based buttons.
        Button btnHourglass = new Button("Hourglass",
                                         new ImageView("hourglass.png"));
        Button btnAnalogClock = new Button("Analog Clock",
                                         new ImageView("analog.png"));

        // Position the text under the image.
        btnHourglass.setContentDisplay(ContentDisplay.TOP);
        btnAnalogClock.setContentDisplay(ContentDisplay.TOP);

        // Handle the action events for the hourglass button.
        btnHourglass.setOnAction(new EventHandler<ActionEvent>() {
            public void handle(ActionEvent ae) {
                response.setText("Hourglass Pressed");
            }
        });
    }
}
```

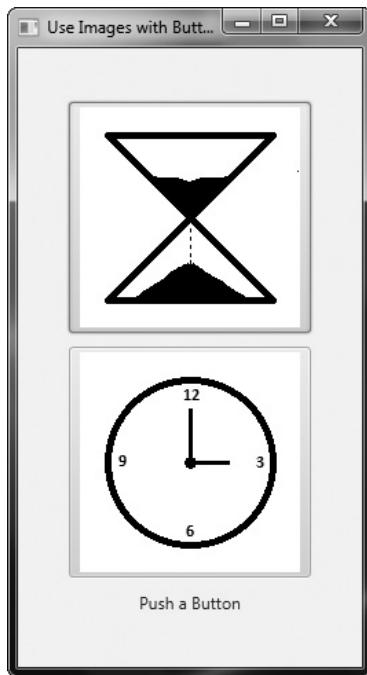
```
// Handle the action events for the analog clock button.  
btnAnalogClock.setOnAction(new EventHandler<ActionEvent>() {  
    public void handle(ActionEvent ae) {  
        response.setText("Analog Clock Pressed");  
    }  
});  
  
// Add the label and buttons to the scene graph.  
rootNode.getChildren().addAll(btnHourglass, btnAnalogClock, response);  
  
// Show the stage and its scene.  
myStage.show();  
}  
}
```

The output produced by this program is shown here:



If you want a button that contains only the image, pass a null string for the text when constructing the button and then call **setContentDisplay()**, passing in the parameter

ContentDisplay.GRAPHIC_ONLY. For example, if you make these modifications to the previous program, the output will look like this:



ToggleButton

A useful variation on the push button is called the *toggle button*. A toggle button looks just like a push button, but it acts differently because it has two states: pushed and released. That is, when you press a toggle button, it stays pressed rather than popping back up as a regular push button does. When you press the toggle button a second time, it releases (pops up). Therefore, each time a toggle button is pushed, it toggles between these two states. In JavaFX, a toggle button is encapsulated in the **ToggleButton** class. Like **Button**, **ToggleButton** is also derived from **ButtonBase**. It implements the **Toggle** interface, which defines functionality common to all types of two-state buttons.

ToggleButton defines three constructors. This is the one we will use:

```
ToggleButton(String str)
```

Here, *str* is the text displayed in the button. Another constructor allows you to include an image. Like other buttons, a **ToggleButton** generates an action event when it is pressed.

Because **ToggleButton** defines a two-state control, it is commonly used to let the user select an option. When the button is pressed, the option is selected. When the button is

released, the option is deselected. For this reason, a program usually needs to determine the toggle button's state. To do this, use the **isSelected()** method, shown here:

```
final boolean isSelected()
```

It returns **true** if the button is pressed and **false** otherwise.

Here is a short program that demonstrates **ToggleButton**:

```
// Demonstrate a toggle button.

import javafx.application.*;
import javafx.scene.*;
import javafx.stage.*;
import javafx.scene.layout.*;
import javafx.scene.control.*;
import javafx.event.*;
import javafx.geometry.*;

public class ToggleButtonDemo extends Application {

    ToggleButton tbOnOff;
    Label response;

    public static void main(String[] args) {

        // Start the JavaFX application by calling launch().
        launch(args);
    }

    // Override the start() method.
    public void start(Stage myStage) {

        // Give the stage a title.
        myStage.setTitle("Demonstrate a Toggle Button");

        // Use a FlowPane for the root node. In this case,
        // vertical and horizontal gaps of 10.
        FlowPane rootNode = new FlowPane(10, 10);

        // Center the controls in the scene.
        rootNode.setAlignment(Pos.CENTER);

        // Create a scene.
        Scene myScene = new Scene(rootNode, 220, 120);

        // Set the scene on the stage.
        myStage.setScene(myScene);

        // Create a label.
        response = new Label("Push the Button.");

        // Create the toggle button.
        tbOnOff = new ToggleButton("On/Off");
    }
}
```

```

// Handle action events for the toggle button.
tbOnOff.setOnAction(new EventHandler<ActionEvent>() {
    public void handle(ActionEvent ae) {
        if(tbOnOff.isSelected()) response.setText("Button is on.");
        else response.setText("Button is off.");
    }
});

// Add the label and buttons to the scene graph.
rootNode.getChildren().addAll(tbOnOff, response);

// Show the stage and its scene.
myStage.show();
}
}

```

Sample output produced by the program is shown here, with the button pressed:

In the program, notice how the pressed/released state of the toggle button is determined by the following lines of code inside the button's action event handler.

```
if(tbOnOff.isSelected()) response.setText("Button is on.");
else response.setText("Button is off.");
```

When the button is pressed, **isSelected()** returns **true**. When the button is released, **isSelected()** returns **false**.

One other point: It is possible to use two or more toggle buttons in a group. In this case, only one button can be in its pressed state at any one time. The process of creating and using a group of toggle buttons is similar to that required to use radio buttons. It is described in the following section.



RadioButton

Another type of button provided by JavaFX is the *radio button*. Radio buttons are a group of mutually exclusive buttons, in which only one button can be selected at any one time. They are supported by the **RadioButton** class, which extends both **ButtonBase** and **ToggleButton**. It also implements the **Toggle** interface. Thus, a radio button is a specialized form of a toggle button. You have almost certainly seen radio buttons in action because they are the primary control employed when the user must select only one option among several alternatives.

To create a radio button, we will use the following constructor:

```
RadioButton(String str)
```

Here, *str* is the label for the button. Like other buttons, when a **RadioButton** is used, an action event is generated.

For their mutually exclusive nature to be activated, radio buttons must be configured into a group. Only one of the buttons in the group can be selected at any time. For example,

if a user presses a radio button that is in a group, any previously selected button in that group is automatically deselected. A button group is created by the **ToggleGroup** class, which is packaged in **javafx.scene.control**. **ToggleGroup** provides only a default constructor.

Radio buttons are added to the toggle group by calling the **setToggleGroup()** method, defined by **ToggleButton**, on the button. It is shown here:

```
final void setToggleGroup(ToggleGroup tg)
```

Here, *tg* is a reference to the toggle button group to which the button is added. After all radio buttons have been added to the same group, their mutually exclusive behavior will be enabled.

In general, when radio buttons are used in a group, one of the buttons is selected when the group is first displayed in the GUI. Here are two ways to do this.

First, you can call **setSelected()** on the button that you want to select. It is defined by **ToggleButton** (which is a superclass of **RadioButton**). It is shown here:

```
final void setSelected(boolean state)
```

If *state* is **true**, the button is selected. Otherwise, it is deselected. Although the button is selected, no action event is generated.

A second way to initially select a radio button is to call **fire()** on the button. It is shown here:

```
void fire()
```

This method results in an action event being generated for the button if the button was previously not selected.

There are a number of different ways to use radio buttons. Perhaps the simplest is to simply respond to the action event that is generated when one is selected. The following program shows an example of this approach. It uses radio buttons to allow the user to select a type of transportation.

```
// A simple demonstration of Radio Buttons.
//
// This program responds to the action events generated
// by a radio button selection. It also shows how to
// fire the button under program control.

import javafx.application.*;
import javafx.scene.*;
import javafx.stage.*;
import javafx.scene.layout.*;
import javafx.scene.control.*;
import javafx.event.*;
import javafx.geometry.*;

public class RadioButtonDemo extends Application {

    Label response;

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

```
// Start the JavaFX application by calling launch().
launch(args);
}

// Override the start() method.
public void start(Stage myStage) {

    // Give the stage a title.
    myStage.setTitle("Demonstrate Radio Buttons");

    // Use a FlowPane for the root node. In this case,
    // vertical and horizontal gaps of 10.
    FlowPane rootNode = new FlowPane(10, 10);

    // Center the controls in the scene.
    rootNode.setAlignment(Pos.CENTER);

    // Create a scene.
    Scene myScene = new Scene(rootNode, 220, 120);

    // Set the scene on the stage.
    myStage.setScene(myScene);

    // Create a label that will report the selection.
    response = new Label("");

    // Create the radio buttons.
    RadioButton rbTrain = new RadioButton("Train");
    RadioButton rbCar = new RadioButton("Car");
    RadioButton rbPlane = new RadioButton("Airplane");

    // Create a toggle group.
    ToggleGroup tg = new ToggleGroup();

    // Add each button to a toggle group.
    rbTrain.setToggleGroup(tg);
    rbCar.setToggleGroup(tg);
    rbPlane.setToggleGroup(tg);

    // Handle action events for the radio buttons.
    rbTrain.setOnAction(new EventHandler<ActionEvent>() {
        public void handle(ActionEvent ae) {
            response.setText("Transport selected is train.");
        }
    });

    rbCar.setOnAction(new EventHandler<ActionEvent>() {
        public void handle(ActionEvent ae) {
            response.setText("Transport selected is car.");
        }
    });

    rbPlane.setOnAction(new EventHandler<ActionEvent>() {
        public void handle(ActionEvent ae) {
            response.setText("Transport selected is airplane.");
        }
    });
}
```

```

        }
    });

    // Fire the event for the first selection. This causes
    // that radio button to be selected and an action event
    // for that button to occur.
    rbTrain.fire();

    // Add the label and buttons to the scene graph.
    rootNode.getChildren().addAll(rbTrain, rbCar, rbPlane, response);

    // Show the stage and its scene.
    myStage.show();
}
}

```

Sample output is shown here:

In the program, pay special attention to how the radio buttons and the toggle group are created. First, the buttons are created using this sequence:

```
RadioButton rbTrain = new RadioButton("Train");
RadioButton rbCar = new RadioButton("Car");
RadioButton rbPlane = new RadioButton("Airplane");
```

Next, a **ToggleGroup** is constructed:

```
ToggleGroup tg = new ToggleGroup();
```

Finally, each radio button is added to the toggle group:

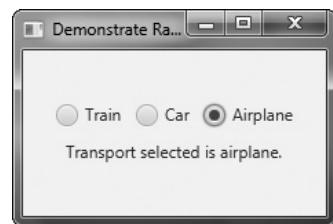
```
rbTrain.setToggleGroup(tg);
rbCar.setToggleGroup(tg);
rbPlane.setToggleGroup(tg);
```

As explained, radio buttons must be part of a toggle group in order for their mutually exclusive behavior to be activated.

After the event handlers for each radio button have been defined, the **rbTrain** button is selected by calling **fire()** on it. This causes that button to be selected and an action event to be generated for it. This causes the button to be initialized with the default selection.

Handling Change Events in a Toggle Group

Although there is nothing wrong, per se, with managing radio buttons by handling action events, as just shown, sometimes it is more appropriate (and easier) to listen to the entire toggle group for changes. When a change takes place, the event handler can easily determine which radio button has been selected and take action accordingly. To use this approach, you must register a **ChangeListener** on the toggle group. When a change event occurs, you



can then determine which button was selected. To try this approach, remove the action event handlers and the call to `fire()` from the preceding program and substitute the following:

```
// Use a change listener to respond to a change of selection within
// the group of radio buttons.
tg.selectedToggleProperty().addListener(new ChangeListener<Toggle>() {
    public void changed(ObservableValue<? extends Toggle> changed,
                        Toggle oldVal, Toggle newVal) {

        // Cast new to RadioButton.
        RadioButton rb = (RadioButton) newVal;

        // Display the selection.
        response.setText("Transport selected is " + rb.getText());
    }
});

// Select the first button. This will cause a change event
// on the toggle group.
rbTrain.setSelected(true);
```

You will also need to add this **import** statement:

```
import javafx.beans.value.*;
```

It supports the **ChangeListener** interface.

The output from this program is the same as before; each time a selection is made, the **response** label is updated. However, in this case, only one event handler is needed for the enter group, rather than three (one for each button). Let's now look at this code more closely.

First, a change event listener is registered for the toggle group. To listen for change events, you must implement the **ChangeListener** interface. This is done by calling `addListener()` on the object returned by `selectedToggleProperty()`. The **ChangeListener** interface defines only one method, called `changed()`. It is shown here:

```
void changed(ObservableValue<? extends T> changed, T oldVal, T newVal)
```

In this case, `changed` is the instance of **ObservableValue<T>**, which encapsulates an object that can be watched for changes. The `oldVal` and `newVal` parameters pass the previous value and the new value, respectively. Thus, in this case, `newVal` holds a reference to the radio button that has just been selected.

In this example, the `setSelected()` method, rather than `fire()`, is called to set the initial selection. Because setting the initial selection causes a change to the toggle group, it results in a change event being generated when the program first begins. You can also use `fire()`, but `setSelected()` was used to demonstrate that any change to the toggle group generates a change event.

An Alternative Way to Handle Radio Buttons

Although handling events generated by radio buttons is often useful, sometimes it is more appropriate to ignore those events and simply obtain the currently selected button when that information is needed. This approach is demonstrated by the following program. It

adds a button called Confirm Transport Selection. When this button is pressed, the currently selected radio button is obtained and then the selected transport is displayed in a label. When you try the program, notice that changing the selected radio button does not cause the confirmed transport to change until you press the Confirm Transport Selection button.

```
// This radio button example demonstrates how the
// currently selected button in a group can be obtained
// under program control, when it is needed, rather
// than responding to action or change events.
//
// In this example, no events related to the radio
// buttons are handled. Instead, the current selection
// is simply obtained when the Confirm Transport Selection push
// button is pressed.

import javafx.application.*;
import javafx.scene.*;
import javafx.stage.*;
import javafx.scene.layout.*;
import javafx.scene.control.*;
import javafx.event.*;
import javafx.geometry.*;

public class RadioButtonDemo2 extends Application {

    Label response;
    ToggleGroup tg;

    public static void main(String[] args) {

        // Start the JavaFX application by calling launch().
        launch(args);
    }

    // Override the start() method.
    public void start(Stage myStage) {

        // Give the stage a title.
        myStage.setTitle("Demonstrate Radio Buttons");

        // Use a FlowPane for the root node. In this case,
        // vertical and horizontal gaps of 10.
        FlowPane rootNode = new FlowPane(10, 10);

        // Center the controls in the scene.
        rootNode.setAlignment(Pos.CENTER);

        // Create a scene.
        Scene myScene = new Scene(rootNode, 200, 140);

        // Set the scene on the stage.
        myStage.setScene(myScene);
    }
}
```

```
// Create two labels.  
Label choose = new Label(" Select a Transport Type ");  
response = new Label("No transport confirmed");  
  
// Create push button used to confirm the selection.  
Button btnConfirm = new Button("Confirm Transport Selection");  
  
// Create the radio buttons.  
RadioButton rbTrain = new RadioButton("Train");  
RadioButton rbCar = new RadioButton("Car");  
RadioButton rbPlane = new RadioButton("Airplane");  
  
// Create a toggle group.  
tg = new ToggleGroup();  
  
// Add each button to a toggle group.  
rbTrain.setToggleGroup(tg);  
rbCar.setToggleGroup(tg);  
rbPlane.setToggleGroup(tg);  
  
// Initially select one of the radio buttons.  
rbTrain.setSelected(true);  
  
// Handle action events for the confirm button.  
btnConfirm.setOnAction(new EventHandler<ActionEvent>() {  
    public void handle(ActionEvent ae) {  
        // Get the radio button that is currently selected.  
        RadioButton rb = (RadioButton) tg.getSelectedToggle();  
  
        // Display the selection.  
        response.setText(rb.getText() + " is confirmed.");  
    }  
});  
  
// Use a separator to better organize the layout.  
Separator separator = new Separator();  
separator.setPrefWidth(180);  
  
// Add the label and buttons to the scene graph.  
rootNode.getChildren().addAll(choose, rbTrain, rbCar, rbPlane,  
    separator, btnConfirm, response);  
  
// Show the stage and its scene.  
myStage.show();  
}  
}
```

The output from the program is shown here:



Most of the program is easy to understand, but two key points are of special interest. First, inside the action event handler for the **btnConfirm** button, notice that the selected radio button is obtained by the following line:

```
RadioButton rb = (RadioButton) tg.getSelectedToggle();
```

Here, the **getSelectedToggle()** method (defined by **ToggleGroup**) obtains the current selection for the toggle group (which, in this case, is a group of radio buttons). It is shown here:

```
final Toggle getSelectedToggle()
```

It returns a reference to the **Toggle** that is selected. In this case, the return value is cast to **RadioButton** because this is the type of button in the group.

The second thing to notice is the use of a visual separator, which is created by this sequence:

```
Separator separator = new Separator();
separator.setPrefWidth(180);
```

The **Separator** class creates a line, which can be either vertical or horizontal. By default, it creates a horizontal line. (A second constructor lets you choose a vertical separator.) **Separator** helps visually organize the layout of controls. It is packaged in **javafx.scene.control**. Next, the width of the separator line is set by calling **setPrefWidth()**, passing in the width.

CheckBox

The **CheckBox** class encapsulates the functionality of a check box. Its immediate superclass is **ButtonBase**. Although you are no doubt familiar with check boxes because they are widely used controls, the JavaFX check box is a bit more sophisticated than you may at first think. This is because **CheckBox** supports three states. The first two are checked or unchecked, as you would expect, and this is the default behavior. The third state is *indeterminate* (also called *undefined*). It is typically used to indicate that the state of the check box has not been set or that it is not relevant to a specific situation. If you need the indeterminate state, you will need to explicitly enable it.

CheckBox defines two constructors. The first is the default constructor. The second lets you specify a string that identifies the box. It is shown here:

```
CheckBox(String str)
```

It creates a check box that has the text specified by *str* as a label. As with other buttons, a **CheckBox** generates an action event when it is selected.

Here is a program that demonstrates check boxes. It displays check boxes that let the user select various deployment options, which are Web, Desktop, and Mobile. Each time a check box state changes, an action event is generated and handled by displaying the new state (selected or cleared) and by displaying a list of all selected boxes.

```
// Demonstrate Check Boxes.
```

```
import javafx.application.*;
import javafx.scene.*;
```

```
import javafx.stage.*;
import javafx.scene.layout.*;
import javafx.scene.control.*;
import javafx.event.*;
import javafx.geometry.*;

public class CheckboxDemo extends Application {

    CheckBox cbWeb;
    CheckBox cbDesktop;
    CheckBox cbMobile;

    Label response;
    Label allTargets;

    String targets = "";

    public static void main(String[] args) {

        // Start the JavaFX application by calling launch().
        launch(args);
    }

    // Override the start() method.
    public void start(Stage myStage) {

        // Give the stage a title.
        myStage.setTitle("Demonstrate Checkboxes");

        // Use a FlowPane for the root node. In this case,
        // vertical and horizontal gaps of 10.
        FlowPane rootNode = new FlowPane(10, 10);

        // Center the controls in the scene.
        rootNode.setAlignment(Pos.CENTER);

        // Create a scene.
        Scene myScene = new Scene(rootNode, 230, 140);

        // Set the scene on the stage.
        myStage.setScene(myScene);

        Label heading = new Label("Select Deployment Options");

        // Create a label that will report the state of the
        // selected check box.
        response = new Label("No Deployment Selected");

        // Create a label that will report all targets selected.
        allTargets = new Label("Target List: <none>");

        // Create the check boxes.
        cbWeb = new CheckBox("Web");
    }
}
```

```
cbDesktop = new CheckBox("Desktop");
cbMobile = new CheckBox("Mobile");

// Handle action events for the check boxes.
cbWeb.setOnAction(new EventHandler<ActionEvent>() {
    public void handle(ActionEvent ae) {
        if(cbWeb.isSelected())
            response.setText("Web deployment selected.");
        else
            response.setText("Web deployment cleared.");

        showAll();
    }
});

cbDesktop.setOnAction(new EventHandler<ActionEvent>() {
    public void handle(ActionEvent ae) {
        if(cbDesktop.isSelected())
            response.setText("Desktop deployment selected.");
        else
            response.setText("Desktop deployment cleared.");

        showAll();
    }
});

cbMobile.setOnAction(new EventHandler<ActionEvent>() {
    public void handle(ActionEvent ae) {
        if(cbMobile.isSelected())
            response.setText("Mobile deployment selected.");
        else
            response.setText("Mobile deployment cleared.");

        showAll();
    }
});

// Use a separator to better organize the layout.
Separator separator = new Separator();
separator.setPrefWidth(200);

// Add controls to the scene graph.
rootNode.getChildren().addAll(heading, separator, cbWeb, cbDesktop,
                             cbMobile, response, allTargets);

// Show the stage and its scene.
myStage.show();
}

// Update and show the targets list.
void showAll() {
    targets = "";
    if(cbWeb.isSelected()) targets = "Web ";
    if(cbDesktop.isSelected()) targets += "Desktop ";
    if(cbMobile.isSelected()) targets += "Mobile ";
    response.setText(targets);
}
```

```

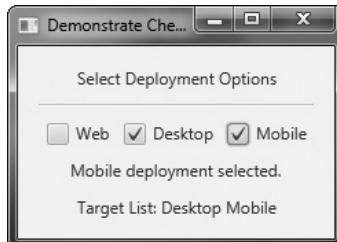
        if (cbDesktop.isSelected()) targets += "Desktop ";
        if (cbMobile.isSelected()) targets += "Mobile";

        if(targets.equals("")) targets = "<none>";

        allTargets.setText("Target List: " + targets);
    }
}

```

Sample output is shown here:



The operation of this program is straightforward. Each time a check box is changed, an action command is generated. To determine if the box is checked or unchecked, the `isSelected()` method is called.

As mentioned, by default, `CheckBox` implements two states: checked and unchecked. If you want to add the indeterminate state, it must be explicitly enabled. To do this, call `setAllowIndeterminate()`, shown here:

```
final void setAllowIndeterminate(boolean enable)
```

In this case, if `enable` is `true`, the indeterminate state is enabled. Otherwise, it is disabled. When the indeterminate state is enabled, the user can select between checked, unchecked, and indeterminate.

You can determine if a check box is in the indeterminate state by calling `isIndeterminate()`, shown here:

```
final boolean isIndeterminate()
```

It returns `true` if the checkbox state is indeterminate and `false` otherwise.

You can see the effect of a three-state check box by modifying the preceding program. First, enable the indeterminate state on the check boxes by calling `setAllowIndeterminate()` on each check box, as shown here:

```

cbWeb.setAllowIndeterminate(true);
cbDesktop.setAllowIndeterminate(true);
cbMobile.setAllowIndeterminate(true);

```

Next, handle the indeterminate state inside the action event handlers. For example, here is the modified handler for `cbWeb`:

```

cbWeb.setOnAction(new EventHandler<ActionEvent>() {
    public void handle(ActionEvent ae) {
        if(cbWeb.isIndeterminate())

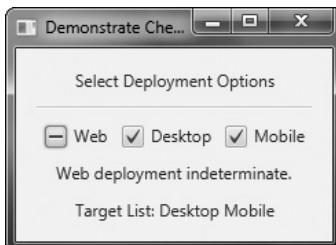
```

```

        response.setText("Web deployment indeterminate.");
    else if(cbWeb.isSelected())
        response.setText("Web deployment selected.");
    else
        response.setText("Web deployment cleared.");

    showAll();
}
);
})
;
```

Now, all three states are tested. Update the other two handlers in the same way. After making these changes, the indeterminate state can be selected, as this sample output shows:



Here, the Web check box is indeterminate.

ListView

Another commonly used control is the list view, which in JavaFX is encapsulated by **ListView**. List views are controls that display a list of entries from which you can select one or more. Because of their ability to make efficient use of limited screen space, list views are popular alternatives to other types of selection controls.

ListView is a generic class that is declared like this:

```
class ListView<T>
```

Here, **T** specifies the type of entries stored in the list view. Often, these are entries of type **String**, but other types are also allowed.

ListView defines two constructors. The first is the default constructor, which creates an empty **ListView**. The second lets you specify the list of entries in the list. It is shown here:

```
ListView(ObservableList<T> list)
```

Here, *list* specifies a list of the items that will be displayed. It is an object of type **ObservableList**, which defines a list of observable objects. It inherits **java.util.List**. Thus, it supports the standard collection methods. **ObservableList** is packaged in **javafx.collections**.

Probably the easiest way to create an **ObservableList** for use in a **ListView** is to use the factory method **observableArrayList()**, which is a static method defined by the **FXCollections** class (which is also packaged in **javafx.collections**). The version we will use is shown here:

```
static <E> ObservableList<E> observableArrayList( E ... elements)
```

In this case, **E** specifies the type of elements, which are passed via *elements*.

By default, a **ListView** allows only one item in the list to be selected at any one time. However, you can allow multiple selections by changing the selection mode. For now, we will use the default, single-selection model.

Although **ListView** provides a default size, sometimes you will want to set the preferred height and/or width to best match your needs. One way to do this is to call the **setPrefHeight()** and **setPrefWidth()** methods, shown here:

```
final void setPrefHeight(double height)  
final void setPrefWidth(double width)
```

Alternatively, you can use a single call to set both dimensions at the same time by use of **setPrefSize()**, shown here:

```
void setPrefSize(double width, double height)
```

There are two basic ways in which you can use a **ListView**. First, you can ignore events generated by the list and simply obtain the selection in the list when your program needs it. Second, you can monitor the list for changes by registering a change listener. This lets you respond each time the user changes a selection in the list. This is the approach used here.

To listen for change events, you must first obtain the selection model used by the **ListView**. This is done by calling **getSelectionModel()** on the list. It is shown here:

```
final MultipleSelectionModel<T> getSelectionModel()
```

It returns a reference to the model. **MultipleSelectionModel** is a class that defines the model used for multiple selections, and it inherits **SelectionModel**. However, multiple selections are allowed in a **ListView** only if multiple-selection mode is turned on.

Using the model returned by **getSelectionModel()**, you will obtain a reference to the selected item property that defines what takes place when an element in the list is selected. This is done by calling **selectedItemProperty()**, shown next:

```
final ReadOnlyObjectProperty<T> selectedItemProperty()
```

You will add the change listener to this property.

The following example puts the preceding discussion into action. It creates a list view that displays various types of transportation, allowing the user to select one. When one is chosen, the selection is displayed.

```
// Demonstrate a list view.  
  
import javafx.application.*;  
import javafx.scene.*;  
import javafx.stage.*;  
import javafx.scene.layout.*;  
import javafx.scene.control.*;  
import javafx.geometry.*;  
import javafx.beans.value.*;  
import javafx.collections.*;  
  
public class ListViewDemo extends Application {  
  
    Label response;
```

```
public static void main(String[] args) {  
    // Start the JavaFX application by calling launch().  
    launch(args);  
}  
  
// Override the start() method.  
public void start(Stage myStage) {  
  
    // Give the stage a title.  
    myStage.setTitle("ListView Demo");  
  
    // Use a FlowPane for the root node. In this case,  
    // vertical and horizontal gaps of 10.  
    FlowPane rootNode = new FlowPane(10, 10);  
  
    // Center the controls in the scene.  
    rootNode.setAlignment(Pos.CENTER);  
  
    // Create a scene.  
    Scene myScene = new Scene(rootNode, 200, 120);  
  
    // Set the scene on the stage.  
    myStage.setScene(myScene);  
  
    // Create a label.  
    response = new Label("Select Transport Type");  
  
    // Create an ObservableList of entries for the list view.  
    ObservableList<String> transportTypes =  
        FXCollections.observableArrayList( "Train", "Car", "Airplane" );  
  
    // Create the list view.  
    ListView<String> lvTransport = new ListView<String>(transportTypes);  
  
    // Set the preferred height and width.  
    lvTransport.setPrefSize(80, 80);  
  
    // Get the list view selection model.  
    MultipleSelectionModel<String> lvSelModel =  
        lvTransport.getSelectionModel();  
  
    // Use a change listener to respond to a change of selection within  
    // a list view.  
    lvSelModel.selectedItemProperty().addListener(  
        new ChangeListener<String>() {  
            public void changed(ObservableValue<? extends String> changed,  
                String oldVal, String newVal) {  
  
                // Display the selection.  
                response.setText("Transport selected is " + newVal);  
            }  
        });  
}
```

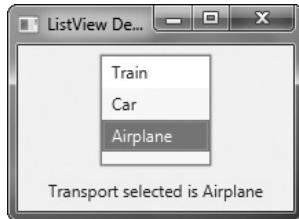
```

        // Add the label and list view to the scene graph.
        rootNode.getChildren().addAll(lvTransport, response);

        // Show the stage and its scene.
        myStage.show();
    }
}

```

Sample output is shown here:



In the program, pay special attention to how the **ListView** is constructed. First, an **ObservableList** is created by this line:

```
ObservableList<String> transportTypes =
    FXCollections.observableArrayList( "Train", "Car", "Airplane" );
```

It uses the **observableArrayList()** method to create a list of strings. Then, the **ObservableList** is used to initialize a **ListView**, as shown here:

```
ListView<String> lvTransport = new ListView<String>(transportTypes);
```

The program then sets the preferred width and height of the control.

Now, notice how the selection model is obtained for **lvTransport**:

```
MultipleSelectionModel<String> lvSelModel =
    lvTransport.getSelectionModel();
```

As explained, **ListView** uses **MultipleSelectionModel**, even when only a single selection is allowed. The **selectedItemProperty()** method is then called on the model and a change listener is registered to the returned item.

ListView Scrollbars

One very useful feature of **ListView** is that when the number of items in the list exceeds the number that can be displayed within its dimensions, scrollbars are automatically added. For example, if you change the declaration of **transportTypes** so that it includes "Bicycle" and "Walking", as shown here:

```
ObservableList<String> transportTypes =
    FXCollections.observableArrayList( "Train", "Car", "Airplane",
        "Bicycle", "Walking" );
```

the **lvTransport** control now looks like the one shown here:



Enabling Multiple Selections

If you want to allow more than one item to be selected, you must explicitly request it. To do so, you must set the selection mode to **SelectionMode.MULTIPLE** by calling **setSelectionMode()** on the **ListView** model. It is shown here:

```
final void setSelectionMode(SelectionMode mode)
```

In this case, *mode* must be either **SelectionMode.MULTIPLE** or **SelectionMode.SINGLE**.

When multiple-selection mode is enabled, you can obtain the list of the selections two ways: as a list of selected indices or as a list of selected items. We will use a list of selected items, but the procedure is similar when using a list of the indices of the selected items. (Note, indexing of items in a **ListView** begins at zero.)

To get a list of the selected items, call **getSelectedItems()** on the selection model. It is shown here:

```
ObservableList<T> getSelectedItems( )
```

It returns an **ObservableList** of the items. Because **ObservableList** extends **java.util.List**, you can access the items in the list just as you would any other **List** collection.

To experiment with multiple selections, you can modify the preceding program as follows. First, make **lvTransport** **final** so it can be accessed within the change event handler. Next, add this line:

```
lvTransport.getSelectionModel().setSelectionMode(SelectionMode.MULTIPLE);
```

It enables multiple-selection mode for **lvTransport**. Finally, replace the change event handler with the one shown here:

```
lvSelModel.selectedItemProperty().addListener(
    new ChangeListener<String>() {
    public void changed(ObservableValue<? extends String> changed,
        String oldVal, String newVal) {

        String selItems = "";
        ObservableList<String> selected =
            lvTransport.getSelectionModel().getSelectedItems();
```

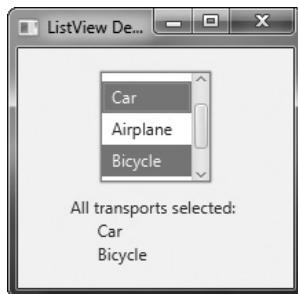
```

// Display the selections.
for(int i=0; i < selected.size(); i++)
    selItems += "\n        " + selected.get(i);

response.setText("All transports selected: " + selItems);
}
});

```

After making these changes, the program will display all selected forms of transports, as the following output shows:



ComboBox

A control related to the list view is the combo box, which is implemented in JavaFX by the **ComboBox** class. A combo box displays one selection, but it will also display a drop-down list that allows the user to select a different item. You can also allow the user to edit a selection. **ComboBox** inherits **ComboBoxBase**, which provides much of its functionality. Unlike the **ListView**, which can allow multiple selections, **ComboBox** is designed for single-selection.

ComboBox is a generic class that is declared like this:

```
class ComboBox<T>
```

Here, **T** specifies the type of entries. Often, these are entries of type **String**, but other types are also allowed.

ComboBox defines two constructors. The first is the default constructor, which creates an empty **ComboBox**. The second lets you specify the list of entries. It is shown here:

```
ComboBox(ObservableList<T> list)
```

In this case, *list* specifies a list of the items that will be displayed. It is an object of type **ObservableList**, which defines a list of observable objects. As explained in the previous section, **ObservableList** inherits **java.util.List**. As also previously explained, an easy way to create an **ObservableList** is to use the factory method **observableArrayList()**, which is a static method defined by the **FXCollections** class.

A **ComboBox** generates an action event when its selection changes. It will also generate a change event. Alternatively, it is also possible to ignore events and simply obtain the current selection when needed.

You can obtain the current selection by calling **getValue()**, shown here:

```
final T getValue()
```

If the value of a combo box has not yet been set (by the user or under program control), then `getValue()` will return `null`. To set the value of a **ComboBox** under program control, call `setValue()`:

```
final void setValue(T newVal)
```

Here, `newVal` becomes the new value.

The following program demonstrates a combo box by reworking the previous list view example. It handles the action event generated by the combo box.

```
// Demonstrate a combo box.

import javafx.application.*;
import javafx.scene.*;
import javafx.stage.*;
import javafx.scene.layout.*;
import javafx.scene.control.*;
import javafx.geometry.*;
import javafx.collections.*;
import javafx.event.*;

public class ComboBoxDemo extends Application {

    ComboBox<String> cbTransport;
    Label response;

    public static void main(String[] args) {

        // Start the JavaFX application by calling launch().
        launch(args);
    }

    // Override the start() method.
    public void start(Stage myStage) {

        // Give the stage a title.
        myStage.setTitle("ComboBox Demo");

        // Use a FlowPane for the root node. In this case,
        // vertical and horizontal gaps of 10.
        FlowPane rootNode = new FlowPane(10, 10);

        // Center the controls in the scene.
        rootNode.setAlignment(Pos.CENTER);

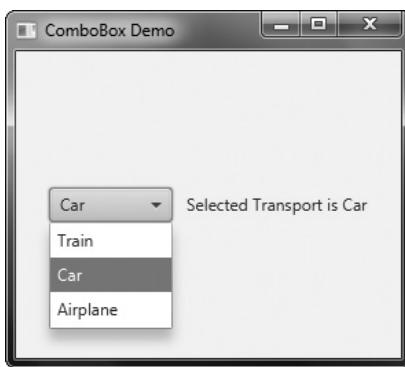
        // Create a scene.
        Scene myScene = new Scene(rootNode, 280, 120);

        // Set the scene on the stage.
        myStage.setScene(myScene);

        // Create a label.
        response = new Label();
    }
}
```

```
// Create an ObservableList of entries for the combo box.  
ObservableList<String> transportTypes =  
    FXCollections.observableArrayList( "Train", "Car", "Airplane" );  
  
// Create a combo box.  
cbTransport = new ComboBox<String>(transportTypes);  
  
// Set the default value.  
cbTransport.setValue("Train");  
  
// Set the response label to indicate the default selection.  
response.setText("Selected Transport is " + cbTransport.getValue());  
  
// Listen for action events on the combo box.  
cbTransport.setOnAction(new EventHandler<ActionEvent>() {  
    public void handle(ActionEvent ae) {  
        response.setText("Selected Transport is " + cbTransport.getValue());  
    }  
});  
  
// Add the label and combo box to the scene graph.  
rootNode.getChildren().addAll(cbTransport, response);  
  
// Show the stage and its scene.  
myStage.show();  
}  
}
```

Sample output is shown here:



As mentioned, **ComboBox** can be configured to allow the user to edit a selection. Assuming that it contains only entries of type **String**, it is easy to enable editing capabilities. Simply call **setEditable()**, shown here:

```
final void setEditable(boolean enable)
```

If *enable* is **true**, editing is enabled. Otherwise, it is disabled. To see the effects of editing, add this line to the preceding program:

```
cbTransport.setEditable(true);
```

After making this addition, you will be able to edit the selection.

ComboBox supports many additional features and functionality beyond those mentioned here. You might find it interesting to explore further. Also, an alternative to a combo box in some cases is the **ChoiceBox** control. You will find it easy to use because it has similarities to both **ListView** and **ComboBox**.

TextField

Although the controls discussed earlier are quite useful and are frequently found in a user interface, they all implement a means of selecting a predetermined option or action. However, sometimes you will want the user to enter a string of his or her own choosing. To accommodate this type of input, JavaFX includes several text-based controls. The one we will look at is **TextField**. It allows one line of text to be entered. Thus, it is useful for obtaining names, ID strings, addresses, and the like. Like all text controls, **TextField** inherits **TextInputControl**, which defines much of its functionality.

TextField defines two constructors. The first is the default constructor, which creates an empty text field that has the default size. The second lets you specify the initial contents of the field. Here, we will use the default constructor.

Although the default size is sometimes adequate, often you will want to specify its size. This is done by calling **setPrefColumnCount()**, shown here:

```
final void setPrefColumnCount(int columns)
```

The *columns* value is used by **TextField** to determine its size.

You can set the text in a text field by calling **setText()**. You can obtain the current text by calling **getText()**. In addition to these fundamental operations, **TextField** supports several other capabilities that you might want to explore, such as cut, paste, and append. You can also select a portion of the text under program control.

One especially useful **TextField** option is the ability to set a prompting message inside the text field when the user attempts to use a blank field. To do this, call **setPromptText()**, shown here:

```
final void setPromptText(String str)
```

In this case, *str* is the string displayed in the text field when no text has been entered. It is displayed using low-intensity (such as a gray tone).

When the user presses ENTER while inside a **TextField**, an action event is generated. Although handling this event is often helpful, in some cases, your program will simply obtain the text when it is needed, rather than handling action events. Both approaches are demonstrated by the following program. It creates a text field that requests a search string. When the user presses ENTER while the text field has input focus, or presses the Get Search String button, the string is obtained and displayed. Notice that a prompting message is also included.

```
// Demonstrate a text field.

import javafx.application.*;
import javafx.scene.*;
import javafx.stage.*;
import javafx.scene.layout.*;
import javafx.scene.control.*;
import javafx.event.*;
import javafx.geometry.*;

public class TextFieldDemo extends Application {

    TextField tf;
    Label response;

    public static void main(String[] args) {

        // Start the JavaFX application by calling launch().
        launch(args);
    }

    // Override the start() method.
    public void start(Stage myStage) {

        // Give the stage a title.
        myStage.setTitle("Demonstrate a TextField");

        // Use a FlowPane for the root node. In this case,
        // vertical and horizontal gaps of 10.
        FlowPane rootNode = new FlowPane(10, 10);

        // Center the controls in the scene.
        rootNode.setAlignment(Pos.CENTER);

        // Create a scene.
        Scene myScene = new Scene(rootNode, 230, 140);

        // Set the scene on the stage.
        myStage.setScene(myScene);

        // Create a label that will report the contents of the
        // text field.
        response = new Label("Search String: ");

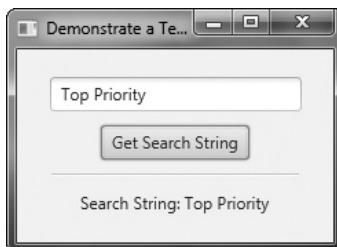
        // Create a button that gets the text.
        Button btnGetText = new Button("Get Search String");

        // Create a text field.
        tf = new TextField();

        // Set the prompt.
        tf.setPromptText("Enter Search String");
    }
}
```

```
// Set preferred column count.  
tf.setPrefColumnCount(15);  
  
// Handle action events for the text field. Action  
// events are generated when ENTER is pressed while  
// the text field has input focus. In this case, the  
// text in the field is obtained and displayed.  
tf.setOnAction(new EventHandler<ActionEvent>() {  
    public void handle(ActionEvent ae) {  
        response.setText("Search String: " + tf.getText());  
    }  
});  
  
// Get text from the text field when the button is pressed  
// and display it.  
btnGetText.setOnAction(new EventHandler<ActionEvent>() {  
    public void handle(ActionEvent ae) {  
        response.setText("Search String: " + tf.getText());  
    }  
});  
  
// Use a separator to better organize the layout.  
Separator separator = new Separator();  
separator.setPrefWidth(180);  
  
// Add controls to the scene graph.  
rootNode.getChildren().addAll(tf, btnGetText, separator, response);  
  
// Show the stage and its scene.  
myStage.show();  
}  
}
```

Sample output is shown here:



Other text controls that you will want to examine include **TextArea**, which supports multiline text, and **PasswordField**, which can be used to input passwords. You might also find **HTMLEditor** helpful.

ScrollPane

Sometimes, the contents of a control will exceed the amount of screen space that you want to give to it. Here are two examples: A large image may not fit within reasonable boundaries, or you might want to display text that is longer than will fit within a small window. Whatever the reason, JavaFX makes it easy to provide scrolling capabilities to any node in a scene graph. This is accomplished by wrapping the node in a **ScrollPane**. When a **ScrollPane** is used, scrollbars are automatically implemented that scroll the contents of the wrapped node. No further action is required on your part. Because of the versatility of **ScrollPane**, you will seldom need to use individual scrollbar controls.

ScrollPane defines two constructors. The first is the default constructor. The second lets you specify a node that you want to scroll. It is shown here:

```
ScrollPane(Node content)
```

In this case, *content* specifies the information to be scrolled. When using the default constructor, you will add the node to be scrolled by calling **setContent()**. It is shown here:

```
final void setContent(Node content)
```

After you have set the content, add the scroll pane to the scene graph. When displayed, the content can be scrolled.

NOTE You can also use **setContent()** to change the content being scrolled by the scroll pane. Thus, what is being scrolled can be changed during the execution of your program.

Although a default size is provided, as a general rule, you will want to set the dimensions of the *viewport*. The viewport is the viewable area of a scroll pane. It is the area in which the content being scrolled is displayed. Thus, the viewport displays the visible portion of the content. The scrollbars scroll the content through the viewport. Thus, by moving a scrollbar, you change what part of the content is visible.

You can set the viewport dimensions by using these two methods:

```
final void setPrefViewportHeight(double height)
```

```
final void setPrefViewportWidth(double width)
```

In its default behavior, a **ScrollPane** will dynamically add or remove a scrollbar as needed. For example, if the component is taller than the viewport, a vertical scrollbar is added. If the component will completely fit within the viewport, the scrollbars are removed.

One nice feature of **ScrollPane** is its ability to pan the contents by dragging the mouse. By default, this feature is off. To turn it on, use **setPannable()**, shown here:

```
final void setPannable(boolean enable)
```

If *enable* is **true**, then panning is allowed. Otherwise, it is disabled.

You can set the position of the scrollbars under program control using **setHvalue()** and **setVvalue()**, shown here:

```
final void setHvalue(double newHval)
```

```
final void setVvalue(double newVval)
```

The new horizontal position is specified by *newHval*, and the new vertical position is specified by *newVval*. By default, scrollbar positions start at zero.

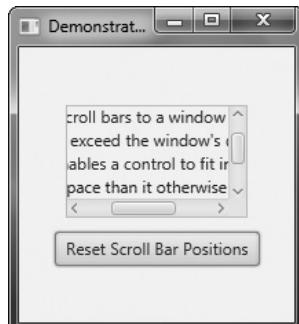
ScrollPane supports various other options. For example, it is possible to set the minimum and maximum scrollbar positions. You can also specify when and if the scrollbars are shown by setting a scrollbar policy. The current position of the scrollbars can be obtained by calling **getHvalue()** and **getVvalue()**.

The following program demonstrates **ScrollPane** by using one to scroll the contents of a multiline label. Notice that it also enables panning.

```
// Demonstrate a scroll pane.  
// This program scrolls the contents of a multiline  
// label, but any node can be scrolled.  
  
import javafx.application.*;  
import javafx.scene.*;  
import javafx.stage.*;  
import javafx.scene.layout.*;  
import javafx.scene.control.*;  
import javafx.event.*;  
import javafx.geometry.*;  
  
public class ScrollPaneDemo extends Application {  
  
    ScrollPane scrlPane;  
  
    public static void main(String[] args) {  
  
        // Start the JavaFX application by calling launch().  
        launch(args);  
    }  
  
    // Override the start() method.  
    public void start(Stage myStage) {  
  
        // Give the stage a title.  
        myStage.setTitle("Demonstrate a ScrollPane");  
  
        // Use a FlowPane for the root node.  
        FlowPane rootNode = new FlowPane(10, 10);  
  
        // Center the controls in the scene.  
        rootNode.setAlignment(Pos.CENTER);  
  
        // Create a scene.  
        Scene myScene = new Scene(rootNode, 200, 200);  
  
        // Set the scene on the stage.  
        myStage.setScene(myScene);  
  
        // Create a label that will be scrolled.  
        Label scrlLabel = new Label(  
            "A ScrollPane streamlines the process of\n" +
```

```
"adding scroll bars to a window whose\n" +
"contents exceed the window's dimensions.\n" +
"It also enables a control to fit in a\n" +
"smaller space than it otherwise would.\n" +
"As such, it often provides a superior\n" +
"approach over using individual scroll bars.");\n\n// Create a scroll pane, setting scrlLabel as the content.\nscrlPane = new ScrollPane(scrlLabel);\n\n// Set the viewport width and height.\nscrlPane.setPrefViewportWidth(130);\nscrlPane.setPrefViewportHeight(80);\n\n// Enable panning.\nscrlPane.setPannable(true);\n\n// Create a reset label.\nButton btnReset = new Button("Reset Scroll Bar Positions");\n\n// Handle action events for the reset button.\nbtnReset.setOnAction(new EventHandler<ActionEvent>() {\n    public void handle(ActionEvent ae) {\n        // Set the scroll bars to their zero position.\n        scrlPane.setVvalue(0);\n        scrlPane.setHvalue(0);\n    }\n});\n\n// Add the label to the scene graph.\nrootNode.getChildren().addAll(scrlPane, btnReset);\n\n// Show the stage and its scene.\nmyStage.show();\n}\n}
```

Sample output is shown here:



TreeView

One of JavaFX's most powerful controls is the **TreeView**. It presents a hierarchical view of data in a tree-like format. In this context, the term *hierarchical* means some items are subordinate to others. For example, a tree is commonly used to display the contents of a file system. In this case, the individual files are subordinate to the directory that contains them. In a **TreeView**, branches can be expanded or collapsed on demand by the user. This allows hierarchical data to be presented in a compact, yet expandable form. Although **TreeView** supports many customization options, you will often find that the default tree style and capabilities are suitable. Therefore, even though trees support a sophisticated structure, they are still quite easy to work with.

TreeView implements a conceptually simple, tree-based data structure. A tree begins with a single *root node* that indicates the start of the tree. Under the root are one or more *child nodes*. There are two types of child nodes: *leaf nodes* (also called *terminal nodes*), which have no children, and *branch nodes*, which form the root nodes of *subtrees*. A subtree is simply a tree that is part of a larger tree. The sequence of nodes that leads from the root to a specific node is called a *path*.

One very useful feature of **TreeView** is that it automatically provides scrollbars when the size of the tree exceeds the dimensions of the view. Although a fully collapsed tree might be quite small, its expanded form may be quite large. By automatically adding scrollbars as needed, **TreeView** lets you use a smaller space than would ordinarily be possible.

TreeView is a generic class that is defined like this:

```
class TreeView<T>
```

Here, **T** specifies the type of value held by an item in the tree. Often, this will be of type **String**. **TreeView** defines two constructors. This is the one we will use:

```
TreeView(TreeItem<T> rootNode)
```

Here, *rootNode* specifies the root of the tree. Because all nodes descend from the root, it is the only one that needs to be passed to **TreeView**.

The items that form the tree are objects of type **TreeItem**. At the outset, it is important to state that **TreeItem** does not inherit **Node**. Thus, **TreeItems** are not general-purpose objects. They can be used in a **TreeView**, but not as stand-alone controls. **TreeItem** is a generic class, as shown here:

```
class TreeItem<T>
```

Here, **T** specifies the type of value held by the **TreeItem**.

Before you can use a **TreeView**, you must construct the tree that it will display. To do this, you must first create the root. Next, add other nodes to that root. You do this by calling either **add()** or **addAll()** on the list returned by **getChildren()**. These other nodes can be leaf nodes or subtrees. After the tree has been constructed, you create the **TreeView** by passing the root node to its constructor.

You can handle selection events in the **TreeView** in a way similar to the way that you handle them in a **ListView**, through the use of a change listener. To do so, first, obtain the selection model by calling **getSelectionModel()**. Then, call **selectedItemProperty()** to obtain the property for the selected item. On that return value, call **addListener()** to add a change listener. Each time a selection is made, a reference to the new selection will be

passed to the **changed()** handler as the new value. (See **ListView** for more details on handling change events.)

You can obtain the value of a **TreeItem** by calling **getValue()**. You can also follow the tree path of an item in either the forward or backward direction. To obtain the parent, call **getParent()**. To obtain the children, call **getChildren()**.

The following example shows how to build and use a **TreeView**. The tree presents a hierarchy of food. The type of items stored in the tree are strings. The root is labeled Food. Under it are three direct descendent nodes: Fruit, Vegetables, and Nuts. Under Fruit are three child nodes: Apples, Pears, and Oranges. Under Apples are three leaf nodes: Fuji, Winesap, and Jonathan. Each time a selection is made, the name of the item is displayed. Also, the path from the root to the item is shown. This is done by the repeated use of **getParent()**.

```
// Demonstrate a TreeView

import javafx.application.*;
import javafx.scene.*;
import javafx.stage.*;
import javafx.scene.layout.*;
import javafx.scene.control.*;
import javafx.event.*;
import javafx.beans.value.*;
import javafx.geometry.*;

public class TreeViewDemo extends Application {

    Label response;

    public static void main(String[] args) {
        // Start the JavaFX application by calling launch().
        launch(args);
    }

    // Override the start() method.
    public void start(Stage myStage) {
        // Give the stage a title.
        myStage.setTitle("Demonstrate a TreeView");

        // Use a FlowPane for the root node. In this case,
        // vertical and horizontal gaps of 10.
        FlowPane rootNode = new FlowPane(10, 10);

        // Center the controls in the scene.
        rootNode.setAlignment(Pos.CENTER);

        // Create a scene.
        Scene myScene = new Scene(rootNode, 310, 460);

        // Set the scene on the stage.
        myStage.setScene(myScene);
    }
}
```

```
// Create a label that will report the state of the
// selected tree item.
response = new Label("No Selection");

// Create tree items, starting with the root.
TreeItem<String> tiRoot = new TreeItem<String>("Food");

// Now add subtrees, beginning with fruit.
TreeItem<String> tiFruit = new TreeItem<String>("Fruit");

// Construct the Apple subtree.
TreeItem<String> tiApples = new TreeItem<String>("Apples");

// Add child nodes to the Apple node.
tiApples.getChildren().add(new TreeItem<String>("Fuji"));
tiApples.getChildren().add(new TreeItem<String>("Winesap"));
tiApples.getChildren().add(new TreeItem<String>("Jonathan"));

// Add varieties to the fruit node.
tiFruit.getChildren().add(tiApples);
tiFruit.getChildren().add(new TreeItem<String>("Pears"));
tiFruit.getChildren().add(new TreeItem<String>("Oranges"));

// Finally, add the fruit node to the root.
tiRoot.getChildren().add(tiFruit);

// Now, add vegetables subtree, using the same general process.
TreeItem<String> tiVegetables = new TreeItem<String>("Vegetables");
tiVegetables.getChildren().add(new TreeItem<String>("Corn"));
tiVegetables.getChildren().add(new TreeItem<String>("Peas"));
tiVegetables.getChildren().add(new TreeItem<String>("Broccoli"));
tiVegetables.getChildren().add(new TreeItem<String>("Beans"));
tiRoot.getChildren().add(tiVegetables);

// Likewise, add nuts subtree.
TreeItem<String> tiNuts = new TreeItem<String>("Nuts");
tiNuts.getChildren().add(new TreeItem<String>("Walnuts"));
tiNuts.getChildren().add(new TreeItem<String>("Peanuts"));
tiNuts.getChildren().add(new TreeItem<String>("Pecans"));
tiRoot.getChildren().add(tiNuts);

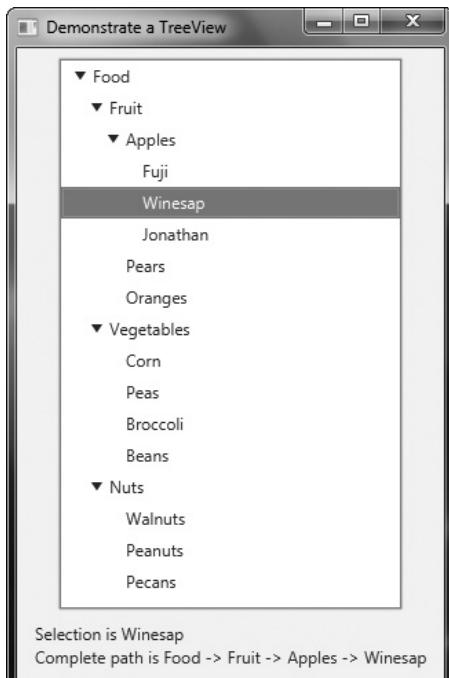
// Create tree view using the tree just created.
TreeView<String> tvFood = new TreeView<String>(tiRoot);

// Get the tree view selection model.
MultipleSelectionModel<TreeItem<String>> tvSelModel =
        tvFood.getSelectionModel();

// Use a change listener to respond to a selection within
// a tree view
tvSelModel.selectedItemProperty().addListener(
        new ChangeListener<TreeItem<String>>() {
    public void changed(
            ObservableValue<? extends TreeItem<String>> changed,
            TreeItem<String> oldVal, TreeItem<String> newVal) {
```

```
// Display the selection and its complete path from the root.  
if(newVal != null) {  
  
    // Construct the entire path to the selected item.  
    String path = newVal.getValue();  
    TreeItem<String> tmp = newVal.getParent();  
    while(tmp != null) {  
        path = tmp.getValue() + " -> " + path;  
        tmp = tmp.getParent();  
    }  
  
    // Display the selection and the entire path.  
    response.setText("Selection is " + newVal.getValue() +  
                     "\nComplete path is " + path);  
}  
}  
});  
  
// Add controls to the scene graph.  
rootNode.getChildren().addAll(tvFood, response);  
  
// Show the stage and its scene.  
myStage.show();  
}  
}  
}
```

Sample output is shown here:



There are two things to pay special attention to in this program. First, notice how the tree is constructed. First, the root node is created by this statement:

```
TreeItem<String> tiRoot = new TreeItem<String>("Food");
```

Next, the nodes under the root are constructed. These nodes consist of the root nodes of subtrees: one for fruit, one for vegetables, and one for nuts. Next, the leaves are added to these subtrees. However, one of these, the fruit subtree, consists of another subtree that contains varieties of apples. The point here is that each branch in a tree leads either to a leaf or to the root of a subtree. After all of the nodes have been constructed, the root nodes of each subtree are added to the root node of the tree. This is done by calling **add()** on the root node. For example, this is how the Nuts subtree is added to **tiRoot**.

```
tiRoot.getChildren().add(tiNuts);
```

The process is the same for adding any child node to its parent node.

The second thing to notice in the program is the way the path from the root to the selected node is constructed within the change event handler. It is shown here:

```
String path = newVal.getValue();
TreeItem<String> tmp = newVal.getParent();
while(tmp != null) {
    path = tmp.getValue() + " -> " + path;
    tmp = tmp.getParent();
}
```

The code works like this: First, the value of the newly selected node is obtained. In this example, the value will be a string, which is the node's name. This string is assigned to the **path** string. Then, a temporary variable of type **TreeItem<String>** is created and initialized to refer to the parent of the newly selected node. If the newly selected node does not have a parent, then **tmp** will be null. Otherwise, the loop is entered, within which each parent's value (which is its name in this case) is added to **path**. This process continues until the root node of the tree (which has no parent) is found.

Although the preceding shows the basic mechanism required to handle a **TreeView**, it is important to point out that several customizations and options are supported. **TreeView** is a powerful control that you will want to examine fully on your own.

Introducing Effects and Transforms

A principal advantage of JavaFX is its ability to alter the precise look of a control (or any node in the scene graph) through the application of an *effect* and/or a *transform*. Both effects and transforms help give your GUI the sophisticated, modern look that users have come to expect. Although it is beyond the scope of this book to examine each effect and transform supported by JavaFX, the following introduction will give you an idea of the benefits they provide.

Effects

Effects are supported by the abstract **Effect** class and its concrete subclasses, which are packaged in **javafx.scene.effect**. Using these effects, you can customize the way a node in a scene graph looks. Several built-in effects are provided. Here is a sampling:

Bloom	Increases the brightness of the brighter parts of a node.
BoxBlur	Blurs a node.
DropShadow	Displays a shadow that appears behind the node.
Glow	Produces a glowing effect.
InnerShadow	Displays a shadow inside a node.
Lighting	Creates shadow effects of a light source.
Reflection	Displays a reflection.

These, and the other effects, are easy to use and are available for use by any **Node**, including controls. Of course, depending on the control, some effects will be more appropriate than others.

To set an effect on a node, call **setEffect()**, which is defined by **Node**. It is shown here:

```
final void setEffect(Effect effect)
```

Here, *effect* is the *effect* that will be applied. To specify no effect, pass **null**. Thus, to add an effect to a node, first create an instance of that effect and then pass it to **setEffect()**. Once this has been done, the effect will be used whenever the node is rendered (as long as the effect is supported by the environment). To demonstrate the power of effects, we will use two of them: **Glow** and **InnerShadow**. However, the process of adding an effect is essentially the same no matter what effect you choose.

Glow produces an effect that gives a node a glowing appearance. The amount of glow is under your control. To use a glow effect, you must first create a **Glow** instance. This is the constructor that we will use:

```
Glow(double glowLevel)
```

Here, *glowLevel* specifies the amount of glowing, which must be a value between 0.0 and 1.0.

After a **Glow** instance has been created, the glow level can be changed by using **setLevel()**, shown here:

```
final void setLevel(double glowLevel)
```

As before, *glowLevel* specifies the glow level, which must be between 0.0 and 1.0.

InnerShadow produces an effect that simulates a shadow on the inside of the node. It supports various constructors. This is the one we will use:

```
InnerShadow(double radius, Color shadowColor)
```

Here, *radius* specifies the radius of the shadow inside the node. In essence, the radius describes the size of the shadow. The color of the shadow is specified by *shadowColor*. Here, the type **Color** is the JavaFX type `javafx.scene.paint.Color`. It defines a large number of constants, such as **Color.GREEN**, **Color.RED**, and **Color.BLUE**, which makes it easy to use.

Transforms

Transforms are supported by the abstract **Transform** class, which is packaged in `javafx.scene.transform`. Four of its subclasses are **Rotate**, **Scale**, **Shear**, and **Translate**. Each does what its name suggests. (Another subclass is **Affine**, but typically you will use one or more of the preceding transform classes.) It is possible to perform more than one transform on a node. For example, you could rotate it and scale it. Transforms are supported by the **Node** class as described next.

One way to add a transform to a node is to add it to the list of transforms maintained by the node. This list is obtained by calling `getTransforms()`, which is defined by **Node**. It is shown here:

```
final ObservableList<Transform> getTransforms()
```

It returns a reference to the list of transforms. To add a transform, simply add it to this list by calling `add()`. You can clear the list by calling `clear()`. You can use `remove()` to remove a specific element.

In some cases, you can specify a transform directly, by setting one of **Node**'s properties. For example, you can set the rotation angle of a node, with the pivot point being at the center of the node, by calling `setRotate()`, passing in the desired angle. You can set a scale by using `setScaleX()` and `setScaleY()`, and you can translate a node by using `setTranslateX()` and `setTranslateY()`. (Z axis translations may also be supported by the platform.) However, using the transforms list offers the greatest flexibility, and that is the approach demonstrated here.

NOTE Any transforms specified on a node directly will be applied after all transforms in the transforms list.

To demonstrate the use of transforms, we will use the **Rotate** and **Scale** classes. The other transforms are used in the same general way. **Rotate** rotates a node around a specified point. It defines several constructors. Here is one example:

```
Rotate(double angle, double x, double y)
```

Here, *angle* specifies the number of degrees to rotate. The center of rotation, called the *pivot point*, is specified by *x* and *y*. It is also possible to use the default constructor and set these values after a **Rotate** object has been created, which is what the following demonstration program will do. This is done by using the `setAngle()`, `setPivotX()`, and `setPivotY()` methods, shown here:

```
final void setAngle(double angle)
final void setPivotX(double x)
final void setPivotY(double y)
```

As before, *angle* specifies the number of degrees to rotate and the center of rotation is specified by *x* and *y*.

Scale scales a node as specified by a scale factor. **Scale** defines several constructors. This is the one we will use:

```
Scale(double widthFactor, double heightFactor)
```

Here, *widthFactor* specifies the scaling factor applied to the node's width, and *heightFactor* specifies the scaling factor applied to the node's height. These factors can be changed after a **Scale** instance has been created by using **setX()** and **setY()**, shown here:

```
final void setX(double widthFactor)  
final void setY(double heightFactor)
```

As before, *widthFactor* specifies the scaling factor applied to the node's width, and *heightFactor* specifies the scaling factor applied to the node's height.

Demonstrating Effects and Transforms

The following program demonstrates the use of effects and transforms. It does so by creating four buttons called Rotate, Scale, Glow, and Shadow. Each time one of these buttons is pressed, the corresponding effect or transform is applied to the button. Sample output is shown here:



When you examine the program, you will see how easy it is to customize the look of your GUI. You might find it interesting to experiment with it, trying different transforms or effects, or trying the effects on different types of nodes other than buttons.

```
// Demonstrate rotation, scaling, glowing, and inner shadow.  
  
import javafx.application.*;  
import javafx.scene.*;  
import javafx.stage.*;  
import javafx.scene.layout.*;  
import javafx.scene.control.*;  
import javafx.event.*;  
import javafx.geometry.*;  
import javafx.scene.transform.*;  
import javafx.scene.effect.*;  
import javafx.scene.paint.*;  
  
public class EffectsAndTransformsDemo extends Application {
```

```
double angle = 0.0;
double glowVal = 0.0;
boolean shadow = false;
double scaleFactor = 1.0;

// Create initial effects and transforms.
Glow glow = new Glow(0.0);
InnerShadow innerShadow = new InnerShadow(10.0, Color.RED);
Rotate rotate = new Rotate();
Scale scale = new Scale(scaleFactor, scaleFactor);

// Create four push buttons.
Button btnRotate = new Button("Rotate");
Button btnGlow = new Button("Glow");
Button btnShadow = new Button("Shadow off");
Button btnScale = new Button("Scale");

public static void main(String[] args) {

    // Start the JavaFX application by calling launch().
    launch(args);
}

// Override the start() method.
public void start(Stage myStage) {

    // Give the stage a title.
    myStage.setTitle("Effects and Transforms Demo");

    // Use a FlowPane for the root node. In this case,
    // vertical and horizontal gaps of 10 are used.
    FlowPane rootNode = new FlowPane(10, 10);

    // Center the controls in the scene.
    rootNode.setAlignment(Pos.CENTER);

    // Create a scene.
    Scene myScene = new Scene(rootNode, 300, 100);

    // Set the scene on the stage.
    myStage.setScene(myScene);

    // Set the initial glow effect.
    btnGlow.setEffect(glow);

    // Add rotation to the transform list for the Rotate button.
    btnRotate.getTransforms().add(rotate);

    // Add scaling to the transform list for the Scale button.
    btnScale.getTransforms().add(scale);

    // Handle the action events for the Rotate button.
    btnRotate.setOnAction(new EventHandler<ActionEvent>() {
```

```
public void handle(ActionEvent ae) {
    // Each time button is pressed, it is rotated 30 degrees
    // around its center.
    angle += 30.0;

    rotate.setAngle(angle);
    rotate.setPivotX(btnRotate.getWidth()/2);
    rotate.setPivotY(btnRotate.getHeight()/2);
}
});

// Handle the action events for the Scale button.
btnScale.setOnAction(new EventHandler<ActionEvent>() {
    public void handle(ActionEvent ae) {
        // Each time button is pressed, the button's scale is changed.
        scaleFactor += 0.1;
        if(scaleFactor > 1.0) scaleFactor = 0.4;

        scale.setX(scaleFactor);
        scale.setY(scaleFactor);

    }
});

// Handle the action events for the Glow button.
btnGlow.setOnAction(new EventHandler<ActionEvent>() {
    public void handle(ActionEvent ae) {
        // Each time button is pressed, its glow value is changed.
        glowVal += 0.1;
        if(glowVal > 1.0) glowVal = 0.0;

        // Set the new glow value.
        glow.setLevel(glowVal);
    }
});

// Handle the action events for the Shadow button.
btnShadow.setOnAction(new EventHandler<ActionEvent>() {
    public void handle(ActionEvent ae) {
        // Each time button is pressed, its shadow status is changed.
        shadow = !shadow;
        if(shadow) {
            btnShadow.setEffect(innerShadow);
            btnShadow.setText("Shadow on");
        } else {
            btnShadow.setEffect(null);
            btnShadow.setText("Shadow off");
        }
    }
});

// Add the label and buttons to the scene graph.
rootNode.getChildren().addAll(btnRotate, btnScale, btnGlow, btnShadow);
```

```

    // Show the stage and its scene.
    myStage.show();
}
}

```

Before leaving the topic of effects and transforms, it is useful to mention that several of them are particularly pleasing when used on a **Text** node. **Text** is a class packaged in **javafx.scene.text**. It creates a node that consists of text. Because it is a node, the text can be easily manipulated as a unit and various effects and transforms can be applied.

Adding Tooltips

One very popular element in the modern GUI is the *tooltip*. A tooltip is a short message that is displayed when the mouse hovers over a control. In JavaFX, a tooltip can be easily added to any control. Frankly, because of the benefits that tooltips offer and the ease by which they can be incorporated into your GUI, there is virtually no reason not to use them where appropriate.

To add a tooltip, you call the **setTooltip()** method defined by **Control**. (**Control** is a base class for all controls.) The **setTooltip()** method is shown here:

```
final void setTooltip(Tooltip tip)
```

In this case, *tip* is an instance of **Tooltip**, which specifies the tooltip. Once a tooltip has been set, it is automatically displayed when the mouse hovers over the control. No other action is required on your part.

The **Tooltip** class encapsulates a tooltip. This is the constructor that we will use:

```
Tooltip(String str)
```

Here, *str* specifies the message that will be displayed by the tooltip.

To see tooltips in action, try adding the following statements to the **CheckboxDemo** program shown earlier.

```

cbWeb.setTooltip(new Tooltip("Deploy to Web"));
cbDesktop.setTooltip(new Tooltip("Deploy to Desktop"));
cbMobile.setTooltip(new Tooltip("Deploy to Mobile"));

```

After these additions, the tooltips will be displayed for each check box.

Disabling a Control

Before leaving the subject of controls, it is useful to describe one more feature. Any node in the scene graph, including a control, can be disabled under program control. To disable a control, use **setDisable()**, defined by **Node**. It is shown here:

```
final void setDisable(boolean disable)
```

If *disable* is **true**, the control is disabled; otherwise, it is enabled. Thus, using **setDisable()**, you can disable a control and then enable it later.

CHAPTER

36

Introducing JavaFX Menus

Menus are an important part of many GUIs because they give the user access to a program's core functionality. Furthermore, the proper implementation of an application's menus is a necessary part of creating a successful GUI. Because of the key role they play in many applications, JavaFX provides extensive support for menus. Fortunately, JavaFX's approach to menus is both powerful and streamlined.

As you will see throughout the course of this chapter, JavaFX menus have several parallels with Swing menus, which were described in Chapter 33. As a result, if you already know how to create Swing menus, learning how to create menus in JavaFX is easy. That said, there are also several differences, so it is important not to jump to conclusions about the JavaFX menu system.

The JavaFX menu system supports several key elements, including

- The menu bar, which is the main menu for an application.
- The standard menu, which can contain either items to be selected or other menus (submenus).
- The context menu, which is often activated by right-clicking the mouse. Context menus are also called popup menus.

JavaFX menus also support *accelerator keys*, which enable menu items to be selected without having to activate the menu, and *mnemonics*, which allow a menu item to be selected by the keyboard once the menu options are displayed. In addition to "normal" menus, JavaFX also supports the *toolbar*, which provides rapid access to program functionality, often paralleling menu items.

Menu Basics

The JavaFX menu system is supported by a group of related classes packaged in **javafx.scene.control**. The ones used in this chapter are shown in Table 36-1, and they represent the core of the menu system. Although JavaFX allows a high degree of

Class	Description
CheckMenuItem	A check menu item.
ContextMenu	A popup menu that is typically activated by right-clicking the mouse.
Menu	A standard menu. A menu consists of one or more MenuItems .
MenuBar	An object that holds the top-level menu for the application.
MenuItem	An object that populates menus.
RadioMenuItem	A radio menu item.
SeparatorMenuItem	A visual separator between menu items.

Table 36-1 The Core JavaFX Menu Classes

customization if desired, normally you will simply use the menu classes as-is because their default look and feel is generally what you will want.

Here is brief overview of how the classes fit together. To create a main menu for an application, you first need an instance of **MenuBar**. This class is, loosely speaking, a container for menus. To the **MenuBar** you add instances of **Menu**. Each **Menu** object defines a menu. That is, each **Menu** object contains one or more selectable items. The items displayed by a **Menu** are objects of type **MenuItem**. Thus, a **MenuItem** defines a selection that can be chosen by the user.

In addition to “standard” menu items, you can also include check and radio menu items in a menu. Their operation parallels check box and radio button controls. A check menu item is created by **CheckMenuItem**. A radio menu item is created by **RadioMenuItem**. Both of these classes extend **MenuItem**.

SeparatorMenuItem is a convenience class that creates a separator line in a menu. It inherits **CustomMenuItem**, which is a class that facilitates embedding other types of controls in a menu item. **CustomMenuItem** extends **MenuItem**.

One key point about JavaFX menus is that **MenuItem** does *not* inherit **Node**. Thus, instances of **MenuItem** can only be used in a menu. They cannot be otherwise incorporated into a scene graph. However, **MenuBar** does inherit **Node**, which does allow the menu bar to be added to the scene graph.

Another key point is that **MenuItem** is a superclass of **Menu**. This allows the creation of submenus, which are, essentially, menus within menus. To create a submenu, you first create and populate a **Menu** object with **MenuItems** and then add it to another **Menu** object. You will see this process in action in the examples that follow.

When a menu item is selected, an action event is generated. The text associated with the selection will be the name of the selection. Thus, when using one action event handler to process all menu selections, one way you can determine which item was selected is by examining the name. Of course, you can also use separate anonymous inner classes or lambda expressions to handle each menu item’s action events. In this case, the menu selection is already known and there is no need to examine the name to determine which item was selected.

As an alternative or adjunct to menus that descend from the menu bar, you can also create stand-alone, context menus, which pop up when activated. To create a context menu, first create an object of type **ContextMenu**. Then, add **MenuItems** to it. A context menu is often activated by clicking the right mouse button when the mouse is over a control

for which a context menu has been defined. It is important to point out that **ContextMenu** is not derived from **MenuItem**. Rather, it inherits **PopupControl**.

A feature related to the menu is the *toolbar*. In JavaFX, toolbars are supported by the **ToolBar** class. It creates a stand-alone component that is often used to provide fast access to functionality contained within the menus of the application. For example, a toolbar might provide fast access to the formatting commands supported by a word processor.

An Overview of **MenuBar**, **Menu**, and **MenuItem**

Before you can create a menu, you need to know some specifics about **MenuBar**, **Menu**, and **MenuItem**. These form the minimum set of classes needed to construct a main menu for an application. **MenuItem**s are also used by context (i.e., popup) menus. Thus, these classes form the foundation of the menu system.

MenuBar

MenuBar is essentially a container for menus. It is the control that supplies the main menu of an application. Like all JavaFX controls, it inherits **Node**. Thus, it can be added to a scene graph. **MenuBar** has only one constructor, which is the default constructor. Therefore, initially, the menu bar will be empty, and you will need to populate it with menus prior to use. As a general rule, an application has one and only one menu bar.

MenuBar defines several methods, but often you will use only one: **getMenus()**. It returns a list of the menus managed by the menu bar. It is to this list that you will add the menus that you create. The **getMenus()** method is shown here:

```
final ObservableList<Menu> getMenus()
```

A **Menu** instance is added to this list of menus by calling **add()**. You can also use **addAll()** to add two or more **Menu** instances in a single call. The added menus are positioned in the bar from left to right, in the order in which they are added. If you want to add a menu at a specific location, then use this version of **add()**:

```
void add(int idx, Menu menu)
```

Here, *menu* is added at the index specified by *idx*. Indexing begins at 0, with 0 being the left-most menu.

In some cases, you might want to remove a menu that is no longer needed. You can do this by calling **remove()** on the **ObservableList** returned by **getMenus()**. Here are two of its forms:

```
void remove(Menu menu)
```

```
void remove(int idx)
```

Here, *menu* is a reference to the menu to remove, and *idx* is the index of the menu to remove. Indexing begins at zero.

It is sometimes useful to obtain a count of the number of items in a menu bar. To do this, call **size()** on the list returned by **getMenus()**.

NOTE Recall that **ObservableList** implements the **List** collections interface, which gives you access to all the methods defined by **List**.

Once a menu bar has been created and populated, it is added to the scene graph in the normal way.

Menu

Menu encapsulates a menu, which is populated with **MenuItem**s. As mentioned, **Menu** is derived from **MenuItem**. This means that one **Menu** can be a selection in another **Menu**. This enables one menu to be submenu of another. **Menu** defines three constructors. Perhaps the most commonly used is shown here:

```
Menu(String name)
```

It creates a menu that has the name specified by *name*. You can specify an image along with text with this constructor:

```
Menu(String name, Node image)
```

Here, *image* specifies the image that is displayed. In all cases, the menu is empty until menu items are added to it. Finally, you don't have to give a menu a name when it is constructed. To create an unnamed menu, you can use the default constructor:

```
Menu()
```

In this case, you can add a name and/or image after the fact by calling **setText()** or **setGraphic()**.

Each menu maintains a list of menu items that it contains. To add an item to the menu, add items to this list. To do so, first call **getItems()**, shown here:

```
final ObservableList<MenuItem> getItems()
```

It returns the list of items currently associated with the menu. To this list, add menu items by calling either **add()** or **addAll()**. Among other actions, you can remove an item by calling **remove()** and obtain the size of the list by calling **size()**.

One other point: You can add a menu separator to the list of menu items, which is an object of type **SeparatorMenuItem**. Separators help organize long menus by allowing you to group related items together. A separator can also help set off an important item, such as the Exit selection in a menu.

MenuItem

MenuItem encapsulates an element in a menu. This element can be either a selection linked to some program action, such as Save or Close, or it can cause a submenu to be displayed. **MenuItem** defines the following three constructors.

```
MenuItem()
```

```
MenuItem(String name)
```

```
MenuItem(String name, Node image)
```

The first creates an empty menu item. The second lets you specify the name of the item, and the third enables you to include an image.

A **MenuItem** generates an action event when selected. You can register an action event handler for such an event by calling **setOnAction()**, just as you did when handling button events. It is shown again for your convenience:

```
final void setOnAction(EventHandler<ActionEvent> handler)
```

Here, *handler* specifies the event handler. You can fire an action event on a menu item by calling **fire()**.

MenuItem defines several methods. One that is often useful is **setDisable()**, which you can use to enable or disable a menu item. It is shown here:

```
final void setDisable(boolean disable)
```

If *disable* is **true**, the menu item is disabled and cannot be selected. If *disable* is **false**, the item is enabled. Using **setDisable()**, you can turn menu items on or off, depending on program conditions.

Create a Main Menu

As a general rule, the most commonly used menu is the *main menu*. This is the menu defined by the menu bar, and it is the menu that defines all (or nearly all) of the functionality of an application. As you will see, JavaFX streamlines the process of creating and managing the main menu. Here, you will see how to construct a simple main menu. Subsequent sections will show various options.

NOTE As a way of clearly illustrating the similarities and differences between the Swing and JavaFX menu systems, the examples in this chapter rework the menu examples from Chapter 33. If you already know Swing, you might find it helpful to compare the two different approaches.

Constructing the main menu requires several steps. First, create the **MenuBar** instance that will hold the menus. Next, construct each menu that will be in the menu bar. In general, a menu is constructed by first creating a **Menu** object and then adding **MenuItems** to it. After the menus have been created, add them to the menu bar. Then, the menu bar, itself, must be added to the scene graph. Finally, for each menu item, you must add an action event handler that responds to the action event fired when a menu item is selected.

A good way to understand the process of creating and managing menus is to work through an example. Here is a program that creates a simple menu bar that contains three menus. The first is a standard File menu that contains Open, Close, Save, and Exit selections. The second menu is called Options, and it contains two submenus called Colors and Priority. The third menu is called Help, and it has one item: About. When a menu item is selected, the name of the selection is displayed in a label.

```
// Demonstrate Menus

import javafx.application.*;
import javafx.scene.*;
import javafx.stage.*;
import javafx.scene.layout.*;
import javafx.scene.control.*;
```

```
import javafx.event.*;
import javafx.geometry.*;

public class MenuDemo extends Application {

    Label response;

    public static void main(String[] args) {

        // Start the JavaFX application by calling launch().
        launch(args);
    }

    // Override the start() method.
    public void start(Stage myStage) {

        // Give the stage a title.
        myStage.setTitle("Demonstrate Menus");

        // Use a BorderPane for the root node.
        BorderPane rootNode = new BorderPane();

        // Create a scene.
        Scene myScene = new Scene(rootNode, 300, 300);

        // Set the scene on the stage.
        myStage.setScene(myScene);

        // Create a label that will report the selection.
        response = new Label("Menu Demo");

        // Create the menu bar.
        MenuBar mb = new MenuBar();

        // Create the File menu.
        Menu fileMenu = new Menu("File");
        MenuItem open = new MenuItem("Open");
        MenuItem close = new MenuItem("Close");
        MenuItem save = new MenuItem("Save");
        MenuItem exit = new MenuItem("Exit");
        fileMenu.getItems().addAll(open, close, save,
                                   new SeparatorMenuItem(), exit);

        // Add File menu to the menu bar.
        mb.getMenus().add(fileMenu);

        // Create the Options menu.
        Menu optionsMenu = new Menu("Options");

        // Create the Colors submenu.
        Menu colorsMenu = new Menu("Colors");
        MenuItem red = new MenuItem("Red");
        MenuItem green = new MenuItem("Green");
        MenuItem blue = new MenuItem("Blue");
```

```
colorsMenu.getItems().addAll(red, green, blue);
optionsMenu.getItems().add(colorsMenu);

// Create the Priority submenu.
Menu priorityMenu = new Menu("Priority");
MenuItem high = new MenuItem("High");
MenuItem low = new MenuItem("Low");
priorityMenu.getItems().addAll(high, low);
optionsMenu.getItems().add(priorityMenu);

// Add a separator.
optionsMenu.getItems().add(new SeparatorMenuItem());

// Create the Reset menu item.
MenuItem reset = new MenuItem("Reset");
optionsMenu.getItems().add(reset);

// Add Options menu to the menu bar.
mb.getMenus().add(optionsMenu);

// Create the Help menu.
Menu helpMenu = new Menu("Help");
MenuItem about = new MenuItem("About");
helpMenu.getItems().add(about);

// Add Help menu to the menu bar.
mb.getMenus().add(helpMenu);

// Create one event handler that will handle menu action events.
EventHandler<ActionEvent> MEHandler =
    new EventHandler<ActionEvent>() {
        public void handle(ActionEvent ae) {
            String name = ((MenuItem)ae.getTarget()).getText();

            // If Exit is chosen, the program is terminated.
            if(name.equals("Exit")) Platform.exit();

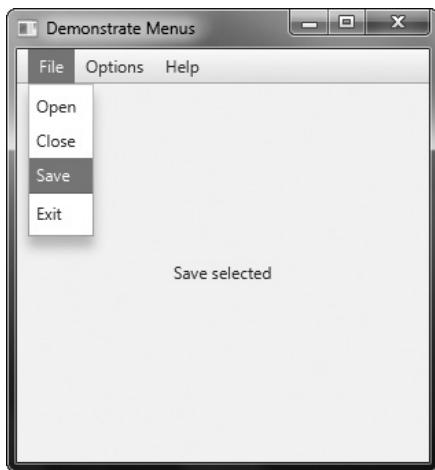
            response.setText( name + " selected");
        }
    };

// Set action event handlers for the menu items.
open.setOnAction(MEHandler);
close.setOnAction(MEHandler);
save.setOnAction(MEHandler);
exit.setOnAction(MEHandler);
red.setOnAction(MEHandler);
green.setOnAction(MEHandler);
blue.setOnAction(MEHandler);
high.setOnAction(MEHandler);
low.setOnAction(MEHandler);
reset.setOnAction(MEHandler);
about.setOnAction(MEHandler);
```

```
// Add the menu bar to the top of the border pane and
// the response label to the center position.
rootNode.setTop(mb);
rootNode.setCenter(response);

// Show the stage and its scene.
myStage.show();
}
}
```

Sample output is shown here:



Let's examine, in detail, how the menus in this program are created. First, note that **MenuDemo** uses a **BorderPane** instance for the root node. **BorderPane** is similar to the AWT's **BorderLayout** discussed in Chapter 26. It defines a window that has five areas: top, bottom, left, right, and center. The following methods set the node assigned to these areas:

```
final void setTop(Node node)
final void setBottom(Node node)
final void setLeft(Node node)
final void setRight(Node node)
final void setCenter(Node node)
```

Here, *node* specifies the element, such as a control, that will be shown in each location. Later in the program, the menu bar is positioned in the top location and a label that displays the menu selection is set to the center position. Setting the menu bar to the top position ensures that it will be shown at the top of the application and will automatically be resized to fit the horizontal width of the window. This is why **BorderPane** is used in the menu examples. Of course, other approaches, such as using a **VBox**, are also valid.

Much of the code in the program is used to construct the menu bar, its menus, and menu items, and this code warrants a close inspection. First, the menu bar is constructed and a reference to it is assigned to **mb** by this statement:

```
// Create the menu bar.
MenuBar mb = new MenuBar();
```

At this point, the menu bar is empty. It will be populated by the menus that follow.

Next, the File menu and its menu entries are created by this sequence:

```
// Create the File menu.
Menu fileMenu = new Menu("File");
MenuItem open = new MenuItem("Open");
MenuItem close = new MenuItem("Close");
MenuItem save = new MenuItem("Save");
MenuItem exit = new MenuItem("Exit");
```

The names Open, Close, Save, and Exit will be shown as selections in the menu. The menu entries are added to the File menu by this call to **addAll()** on the list of menu items returned by **getItems()**:

```
fileMenu.getItems().addAll(open, close, save,
                           new SeparatorMenuItem(), exit);
```

Recall that **getItems()** returns the menu items associated with a **Menu** instance. To add menu items to a menu, you will add them to this list. Notice that a separator is used to separate visually the Exit entry from the others.

Finally, the File menu is added to the menu bar by this line:

```
// Add File menu to the menu bar.
mb.getMenus().add(fileMenu);
```

Once the preceding code sequence completes, the menu bar will contain one entry: File. The File menu will contain four selections in this order: Open, Close, Save, and Exit.

The Options menu is constructed using the same basic process as the File menu. However, the Options menu consists of two submenus, Colors and Priority, and a Reset entry. The submenus are first constructed individually and then added to the Options menu. As explained, because **Menu** inherits **MenuItem**, a **Menu** can be added as an entry into another **Menu**. This is the way the submenus are created. The Reset item is added last. Then, the Options menu is added to the menu bar. The Help menu is constructed using the same process.

After all of the menus have been constructed, an **ActionEvent** handler called **MEHandler** is created that will process menu selections. For demonstration purposes, a single handler will process all selections, but in a real-world application, it is often easier to specify a separate handler for each individual selection by using anonymous inner classes or lambda expressions. The **ActionEvent** handler for the menu items is shown here:

```
// Create one event handler that will handle all menu events.
EventHandler<ActionEvent> MEHandler = new EventHandler<ActionEvent>() {
```

```

public void handle(ActionEvent ae) {
    String name = ((MenuItem)ae.getTarget()).getText();

    // If Exit is chosen, the program is terminated.
    if(name.equals("Exit")) Platform.exit();

    response.setText( name + " selected");
}
};

```

Inside **handle()**, the target of the event is obtained by calling **getTarget()**. The returned reference is cast to **MenuItem**, and its name is returned by calling **getText()**. This string is then assigned to **name**. If **name** contains the string "Exit", the application is terminated by calling **Platform.exit()**. Otherwise, the name is displayed in the **response** label.

Before continuing, it must be pointed out that a JavaFX application must call **Platform.exit()**, not **System.exit()**. The **Platform** class is defined by JavaFX and packaged in **javafx.application**. Its **exit()** method causes the **stop()** life-cycle method to be called. **System.exit()** does not.

Finally, **MEHandler** is registered as the action event handler for each menu item by the following statements:

```

// Set action event handlers for the menu items.
open.setOnAction(MEHandler);
close.setOnAction(MEHandler);
save.setOnAction(MEHandler);
exit.setOnAction(MEHandler);
red.setOnAction(MEHandler);
green.setOnAction(MEHandler);
blue.setOnAction(MEHandler);
high.setOnAction(MEHandler);
low.setOnAction(MEHandler);
reset.setOnAction(MEHandler);
about.setOnAction(MEHandler);

```

Notice that no listeners are added to the Colors or Priority items because they are not actually selections. They simply activate submenus.

Finally, the menu bar is added to the root node by the following line:

```
rootNode.setTop(mb);
```

This causes the menu bar to be placed at the top of the window.

At this point, you might want to experiment a bit with the **MenuDemo** program. Try adding another menu or adding additional items to an existing menu. It is important that you understand the basic menu concepts before moving on because this program will evolve throughout the remainder of this chapter.

Add Mnemonics and Accelerators to Menu Items

The menu created in the preceding example is functional, but it is possible to make it better. In real applications, a menu usually includes support for keyboard shortcuts. These come in two forms: accelerators and mnemonics. An accelerator is a key combination that

lets you select a menu item without having to first activate the menu. As it applies to menus, a mnemonic defines a key that lets you select an item from an active menu by typing the key. Thus, a mnemonic allows you to use the keyboard to select an item from a menu that is already being displayed.

An accelerator can be associated with a **Menu** or **MenuItem**. It is specified by calling **setAccelerator()**, shown next:

```
final void setAccelerator(KeyCombination keyComb)
```

Here, *keyComb* is the key combination that is pressed to select the menu item.

KeyCombination is class that encapsulates a key combination, such as CTRL-S. It is packaged in **javafx.scene.input**.

KeyCombination defines two **protected** constructors, but often you will use the **keyCombination()** factory method, shown here:

```
static KeyCombination keyCombination(String keys)
```

In this case, *keys* is a string that specifies the key combination. It typically consists of a modifier, such as CTRL, ALT, SHIFT, or META, and a letter, such as S. There is a special value, called **shortcut**, which can be used to specify the CTRL key in a Windows system and the META key on a Mac. (It also maps to the typically used shortcut key on other types of systems.) Therefore, if you want to specify CTRL-S as the key combination for Save, then use the string "shortcut+S". This way, it will work for both Windows and Mac and elsewhere.

The following sequence adds accelerators to the File menu created by the **MenuDemo** program in the previous section. After making this change, you can directly select a File menu option by pressing CTRL-O, CTRL-C, CTRL-S, or CTRL-E.

```
// Add keyboard accelerators for the File menu.
open.setAccelerator(KeyCombination.keyCombination("shortcut+O"));
close.setAccelerator(KeyCombination.keyCombination("shortcut+C"));
save.setAccelerator(KeyCombination.keyCombination("shortcut+S"));
exit.setAccelerator(KeyCombination.keyCombination("shortcut+E"));
```

A mnemonic can be specified for both **MenuItem** and **Menu** objects, and it is very easy to do. Simply precede the letter in the name of the menu or menu item with an underscore. For example, in the preceding example, to add the mnemonic *F* to the File menu, declare **fileMenu** as shown here:

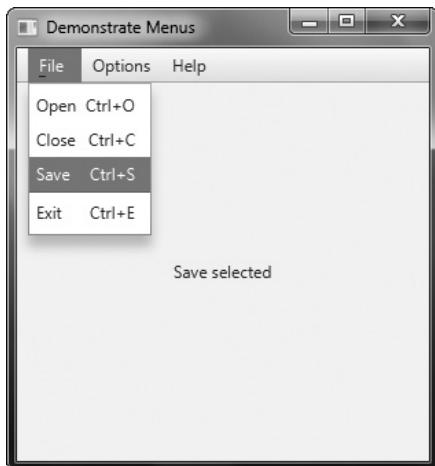
```
Menu fileMenu = new Menu("_File"); // now defines a mnemonic
```

After making this change, you can select the File menu by typing ALT then F. However, mnemonics are active only if mnemonic parsing is **true** (as it is by default). You can turn mnemonic parsing on or off by using **setMnemonicParsing()**, shown here:

```
final void setMnemonicParsing(boolean enable)
```

In this case, if *enable* is **true**, then mnemonic parsing is turned on. Otherwise, it is turned off.

After making these changes, the File menu will now look like this:



Add Images to Menu Items

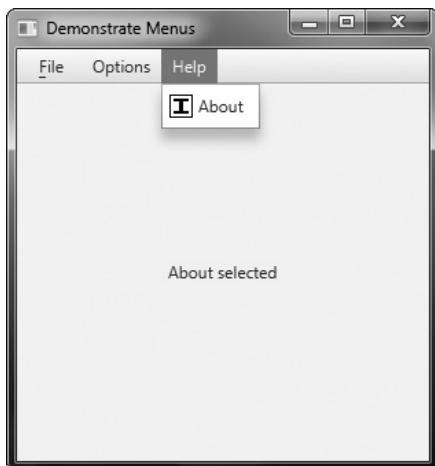
You can add images to menu items or use images instead of text. The easiest way to add an image is to specify it when the menu item is being constructed using this constructor:

```
MenuItem(String name, Node image)
```

It creates a menu item with the name specified by *name* and the image specified by *image*. For example, here the About menu item is associated with an image when it is created.

```
ImageView aboutIV = new ImageView("aboutIcon.gif");  
MenuItem about = new MenuItem("About", aboutIV);
```

After this addition, the image specified by **aboutIV** will be displayed next to the text "About" when the Help menu is displayed, as shown here:



One last point: You can also add an image to a menu item after the item has been created by calling `setGraphic()`. This lets you change the image during program execution.

Use RadioMenuItem and CheckMenuItem

Although the type of menu items used by the preceding examples are, as a general rule, the most commonly used, JavaFX defines two others: check menu items and radio menu items. These elements can streamline a GUI by allowing a menu to provide functionality that would otherwise require additional, stand-alone components. Also, sometimes including check or radio menu items simply seems most natural for a specific set of features. Whatever your reason, it is easy to use check and/or radio menu items in menus, and both are examined here.

To add a check menu item to a menu, use **CheckMenuItem**. It defines three constructors, which parallel the ones defined by **MenuItem**. The one used in this chapter is shown here:

```
CheckMenuItem(String name)
```

Here, *name* specifies the name of the item. The initial state of the item is unchecked. If you want to check a check menu item under program control, call `setSelected()`, shown here:

```
final void setSelected(boolean selected)
```

If *selected* is **true**, the menu item is checked. Otherwise, it is unchecked.

Like stand-alone check boxes, check menu items generate action events when their state is changed. Check menu items are especially appropriate in menus when you have options that can be selected and you want to display their selected/deselected status.

A radio menu item can be added to a menu by creating an object of type **RadioMenuItem**. **RadioMenuItem** defines a number of constructors. The one used in this chapter is shown here:

```
RadioMenuItem(String name)
```

It creates a radio menu item that has the name passed in *name*. The item is not selected. As with the case of check menu items, to select a radio menu item, call `setSelected()`, passing **true** as an argument.

RadioMenuItem works like a stand-alone radio button, generating both change and action events. Like stand-alone radio buttons, menu radio items must be put into a toggle group in order for them to exhibit mutually exclusive selection behavior.

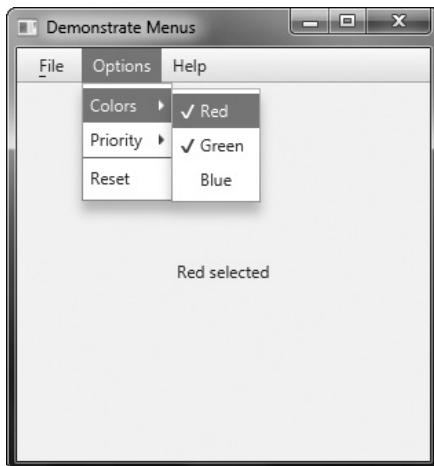
Because both **CheckMenuItem** and **RadioMenuItem** inherit **MenuItem**, each has all of the functionality provided by **MenuItem**. Aside from having the extra capabilities of check boxes and radio buttons, they act like and are used like other menu items.

To try check and radio menu items, first remove the code that creates the Options menu in the **MenuDemo** example program. Then substitute the following code sequence, which uses check menu items for the Colors submenu and radio menu items for the Priority submenu.

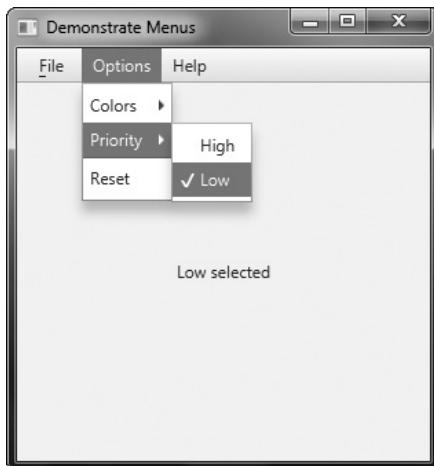
```
// Create the Options menu.  
Menu optionsMenu = new Menu("Options");
```

```
// Create the Colors submenu.  
Menu colorsMenu = new Menu("Colors");  
  
// Use check menu items for colors. This allows  
// the user to select more than one color.  
CheckMenuItem red = new CheckMenuItem("Red");  
CheckMenuItem green = new CheckMenuItem("Green");  
CheckMenuItem blue = new CheckMenuItem("Blue");  
colorsMenu.getItems().addAll(red, green, blue);  
optionsMenu.getItems().add(colorsMenu);  
  
// Select green for the default color selection.  
green.setSelected(true);  
  
// Create the Priority submenu.  
Menu priorityMenu = new Menu("Priority");  
  
// Use radio menu items for the priority setting.  
// This lets the menu show which priority is used  
// and also ensures that one and only one priority  
// can be selected at any one time.  
RadioMenuItem high = new RadioMenuItem("High");  
RadioMenuItem low = new RadioMenuItem("Low");  
  
// Create a toggle group and use it for the radio menu items.  
ToggleGroup tg = new ToggleGroup();  
high.setToggleGroup(tg);  
low.setToggleGroup(tg);  
  
// Select High priority for the default selection.  
high.setSelected(true);  
  
// Add the radio menu items to the Priority menu and  
// add the Priority menu to the Options menu.  
priorityMenu.getItems().addAll(high, low);  
optionsMenu.getItems().add(priorityMenu);  
  
// Add a separator.  
optionsMenu.getItems().add(new SeparatorMenuItem());  
  
// Create the Reset menu item.  
MenuItem reset = new MenuItem("Reset");  
optionsMenu.getItems().add(reset);  
  
// Add Options menu to the menu bar.  
mb.getMenus().add(optionsMenu);
```

After making the substitution, the check menu items in the Colors submenu look like those shown here:



Here is how the radio menu items in the Priority submenu now look:



Create a Context Menu

A popular alternative or addition to the menu bar is the popup menu, which in JavaFX is referred to as a *context menu*. Typically, a context menu is activated by clicking the right mouse button when over a control. Popup menus are supported in JavaFX by the **ContextMenu** class. The direct superclass of **ContextMenu** is **PopupControl**. An indirect superclass of **ContextMenu** is **javafx.stage.PopupWindow**, which supplies much of its basic functionality.

ContextMenu has two constructors. The one used in this chapter is shown here:

```
ContextMenu(MenuItem ... menuItems)
```

Here, *menuItems* specify the menu items that will constitute the context menu. The second **ContextMenu** constructor creates an empty menu to which items must be added.

In general, context menus are constructed like regular menus. Menu items are created and added to the menu. Menu item selections are also handled in the same way: by handling action events. The main difference between a context menu and a regular menu is the activation process.

To associate a context menu with a control is amazingly easy. Simply call **setContextMenu()** on the control, passing in a reference to the menu that you want to pop up. When you right-click on that control, the associated context menu will be shown. The **setContextMenu()** method is shown here:

```
final void setContextMenu(ContextMenu menu)
```

In this case, *menu* specifies the context menu associated with the invoking control.

To demonstrate a context menu, we will add one to the **MenuDemo** program. The context menu will present a standard “Edit” menu that includes the Cut, Copy, and Paste entries. It will be set on a text field control. When the mouse is right-clicked while in the text field, the context menu will pop up. To begin, create the context menu, as shown here:

```
// Create the context menu items
MenuItem cut = new MenuItem("Cut");
MenuItem copy = new MenuItem("Copy");
MenuItem paste = new MenuItem("Paste");

// Create a context (i.e., popup) menu that shows edit options.
final ContextMenu editMenu = new ContextMenu(cut, copy, paste);
```

This sequence begins by constructing the **MenuItem**s that will form the menu. It then creates an instance of **ContextMenu** called **editMenu** that contains the items.

Next, add the action event handler to these menu items, as shown here:

```
cut.setOnAction(MEHandler);
copy.setOnAction(MEHandler);
paste.setOnAction(MEHandler);
```

This finishes the construction of the context menu, but the menu has not yet been associated with a control.

Now, add the following sequence that creates the text field:

```
// Create a text field and set its column width to 20.
TextField tf = new TextField();
tf.setPrefColumnCount(20);
```

Next, set the context menu on the text field:

```
// Add the context menu to the textfield.
tf.setContextMenu(editMenu);
```

Now, when the mouse is right-clicked over the text field, the context menu will pop up.

To add the text field to the program, you must create a flow pane that will hold both the text field and the response label. This pane will then be added to the center of the **BorderPane**. This step is necessary because only one node can be added to any single location within a **BorderPane**. First, remove this line of code:

```
rootNode.setCenter(response);
```

Replace it with the following code:

```
// Create a flow pane that will hold both the response
// label and the text field.
FlowPane fpRoot = new FlowPane(10, 10);

// Center the controls in the scene.
fpRoot.setAlignment(Pos.CENTER);

// Add both the label and the text field to the flow pane.
fpRoot.getChildren().addAll(response, tf);

// Add the flow pane to the center of the border layout.
rootNode.setCenter(fpRoot);
```

Of course, the menu bar is still added to the top position of the border pane.

After making these changes, when you right-click over the text field, the context menu will pop up, as shown here:



It is also possible to associate a context menu with a scene. One way to do this is by calling **setOnContextMenuRequested()** on the root node of the scene. This method is defined by **Node** and is shown here:

```
final void setOnContextMenuRequested(
    EventHandler<? super ContextMenuEvent> eventHandler)
```

Here, *eventHandler* specifies the handler that will be called when a popup request has been received for the context menu. In this case, the handler must call the **show()** method defined by **ContextMenu** to cause the context menu to be displayed. This is the version we will use:

```
final void show(Node node, double upperX, double upperY)
```

Here, *node* is the element on which the context menu is linked. The values of *upperX* and *upperY* define the X,Y location of the upper-left corner of the menu, relative to the screen. Typically, you will pass the screen coordinates at which the right-click occurred. To do this, you will call the **getScreenX()** and **getScreenY()** methods defined by **ContextMenuEvent**. They are shown here:

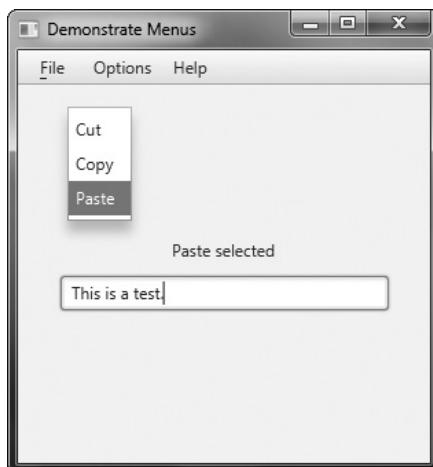
```
final double getScreenX()
final double getScreenY()
```

Thus, you will typically pass the results of these methods to the **show()** method.

The preceding theory can be put into practice by adding the context menu to the root node of the scene graph. After doing so, right-clicking anywhere in the scene will cause the menu to pop up. To do this, first add the following sequence to the **MenuDemo** program:

```
// Add the context menu to the entire scene graph.
rootNode.setOnContextMenuRequested(
    new EventHandler<ContextMenuEvent>() {
        public void handle(ContextMenuEvent ae) {
            // Popup menu at the location of the right click.
            editMenu.show(rootNode, ae.getScreenX(), ae.getScreenY());
        }
});
```

Second, declare **rootNode** **final** so that it can be accessed within the anonymous inner class. After you have made these additions and changes, the context menu can be activated by clicking the right mouse button anywhere inside the application scene. For example, here is the menu displayed after right-clicking in the upper-left portion of the window.



Create a Toolbar

A toolbar is a component that can serve as both an alternative and as an adjunct to a menu. Typically, a toolbar contains a list of buttons that give the user immediate access to various program options. For example, a toolbar might contain buttons that select various font options, such as bold, italics, highlight, or underline. These options can be selected without the need to drop through a menu. As a general rule, toolbar buttons show images rather than text, although either or both are allowed. Furthermore, often tooltips are associated with image-based toolbar buttons.

In JavaFX, toolbars are instances of the **ToolBar** class. It defines the two constructors, shown here:

```
ToolBar()
ToolBar(Node ... nodes)
```

The first constructor creates an empty, horizontal toolbar. The second creates a horizontal toolbar that contains the specified nodes, which are usually some form of button. If you want to create a vertical toolbar, call **setOrientation()** on the toolbar. It is shown here:

```
final void setOrientation(Orientation how)
```

The value of *how* must be either **Orientation.VERTICAL** or **Orientation.HORIZONTAL**.

You add buttons (or other controls) to a toolbar in much the same way that you add them to a menu bar: call **add()** on the reference returned by the **getItems()** method. Often, however, it is easier to specify the items in the **ToolBar** constructor, and that is the approach used in this chapter. Once you have created a toolbar, add it to the scene graph. For example, when using a border layout, it could be added to the bottom location. Of course, other approaches are commonly used. For example, it could be added to a location directly under the menu bar or at the side of the window.

To illustrate a toolbar, we will add one to the **MenuDemo** program. The toolbar will present three debugging options: Set Breakpoint, Clear Breakpoint, and Resume Execution. We will also add tooltips to the menu items. Recall from the previous chapter, a tooltip is a small message that describes an item. It is automatically displayed if the mouse hovers over the item for moment. You can add a tooltip to the menu item in the same way as you add it to a control: by calling **setTooltip()**. Tooltips are especially useful when applied to image-based toolbar controls because sometimes it's hard to design images that are intuitive to all users.

First, add the following code, which creates the debugging toolbar:

```
// Define a toolbar. First, create toolbar items.
Button btnSet = new Button("Set Breakpoint",
                           new ImageView("setBP.gif"));
Button btnClear = new Button("Clear Breakpoint",
                            new ImageView("clearBP.gif"));
Button btnResume = new Button("Resume Execution",
                             new ImageView("resume.gif"));

// Now, turn off text in the buttons.
btnSet.setContentDisplay(ContentDisplay.GRAPHIC_ONLY);
btnClear.setContentDisplay(ContentDisplay.GRAPHIC_ONLY);
btnResume.setContentDisplay(ContentDisplay.GRAPHIC_ONLY);
```

```
// Set tooltips.
btnSet.setTooltip(new Tooltip("Set a breakpoint."));
btnClear.setTooltip(new Tooltip("Clear a breakpoint."));
btnResume.setTooltip(new Tooltip("Resume execution."));

// Create the toolbar.
ToolBar tbDebug = new ToolBar(btnSet, btnClear, btnResume);
```

Let's look at this code closely. First, three buttons are created that correspond to the debug actions. Notice that each has an image associated with it. Next, each button deactivates the text display by calling **setContentDisplay()**. As a point of interest, it would have been possible to leave the text displayed, but the toolbar would have had a somewhat nonstandard look. (The text for each button is still needed, however, because it will be used by the action event handler for the buttons.) Tooltips are then set for each button. Finally, the toolbar is created, with the buttons specified as the contents.

Next, add the following sequence, which defines an action event handler for the toolbar buttons:

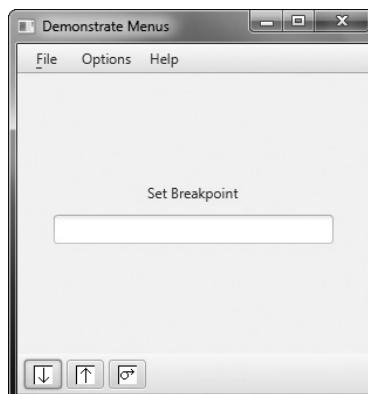
```
// Create a handler for the toolbar buttons.
EventHandler<ActionEvent> btnHandler = new EventHandler<ActionEvent>() {
    public void handle(ActionEvent ae) {
        response.setText(((Button)ae.getTarget()).getText());
    }
};

// Set the toolbar button action event handlers.
btnSet.setOnAction(btnHandler);
btnClear.setOnAction(btnHandler);
btnResume.setOnAction(btnHandler);
```

Finally, add the toolbar to the bottom of the border layout by using this statement:

```
rootNode.setBottom(tbDebug);
```

After making these additions, each time the user presses a toolbar button, an action event is fired, and it is handled by displaying the button's text in the **response** label. The following output shows the toolbar in action.



Put the Entire **MenuDemo** Program Together

Throughout the course of this discussion, many changes and additions have been made to the **MenuDemo** program shown at the start of the chapter. Before concluding, it will be helpful to assemble all the pieces. Doing so not only eliminates any ambiguity about the way the pieces fit together, but it also gives you a complete menu demonstration program that you can experiment with.

The following version of **MenuDemo** includes all of the additions and enhancements described in this chapter. For clarity, the program has been reorganized, with separate methods being used to construct the various menus and toolbar. Notice that several of the menu-related variables, such as **mb** and **tbDebug**, have been made into instance variables so they can be directly accessed by any part of the class.

```
// Demonstrate Menus -- Final Version

import javafx.application.*;
import javafx.scene.*;
import javafx.stage.*;
import javafx.scene.layout.*;
import javafx.scene.control.*;
import javafx.event.*;
import javafx.geometry.*;
import javafx.scene.input.*;
import javafx.scene.image.*;
import javafx.beans.value.*;

public class MenuDemoFinal extends Application {

    MenuBar mb;
    EventHandler<ActionEvent> MEHandler;
    ContextMenu editMenu;
    ToolBar tbDebug;

    Label response;

    public static void main(String[] args) {
        // Start the JavaFX application by calling launch().
        launch(args);
    }

    // Override the start() method.
    public void start(Stage myStage) {
        // Give the stage a title.
        myStage.setTitle("Demonstrate Menus -- Final Version");

        // Use a BorderPane for the root node.
        final BorderPane rootNode = new BorderPane();

        // Create a scene.
        Scene myScene = new Scene(rootNode, 300, 300);
```

```
// Set the scene on the stage.  
myStage.setScene(myScene);  
  
// Create a label that will report the selection.  
response = new Label();  
  
// Create one event handler for all menu action events.  
MEHandler = new EventHandler<ActionEvent>() {  
    public void handle(ActionEvent ae) {  
        String name = ((MenuItem)ae.getTarget()).getText();  
  
        if(name.equals("Exit")) Platform.exit();  
  
        response.setText( name + " selected");  
    }  
};  
  
// Create the menu bar.  
mb = newMenuBar();  
  
// Create the File menu.  
makeFileMenu();  
  
// Create the Options menu.  
makeOptionsMenu();  
  
// Create the Help menu.  
makeHelpMenu();  
  
// Create the context menu.  
makeContextMenu();  
  
// Create a text field and set its column width to 20.  
TextField tf = new TextField();  
tf.setPrefColumnCount(20);  
  
// Add the context menu to the text field.  
tf.setContextMenu(editMenu);  
  
// Create the toolbar.  
makeToolBar();  
  
// Add the context menu to the entire scene graph.  
rootNode.setOnContextMenuRequested(  
    new EventHandler<ContextMenuEvent>() {  
        public void handle(ContextMenuEvent ae) {  
            // Popup menu at the location of the right click.  
            editMenu.show(rootNode, ae.getScreenX(), ae.getScreenY());  
        }  
});  
  
// Add the menu bar to the top of the border pane.  
rootNode.setTop(mb);
```

```
// Create a flow pane that will hold both the response
// label and the text field.
FlowPane fpRoot = new FlowPane(10, 10);

// Center the controls in the scene.
fpRoot.setAlignment(Pos.CENTER);

// Use a separator to better organize the layout.
Separator separator = new Separator();
separator.setPrefWidth(260);

// Add the label, separator, and text field to the flow pane.
fpRoot.getChildren().addAll(response, separator, tf);

// Add the toolbar to the bottom of the border pane.
rootNode.setBottom(tbDebug);

// Add the flow pane to the center of the border layout.
rootNode.setCenter(fpRoot);

// Show the stage and its scene.
myStage.show();
}

// Create the File menu.
void makeFileMenu() {
    // Create the File menu, including a mnemonic.
    Menu fileMenu = new Menu("_File");

    // Create the File menu items.
    MenuItem open = new MenuItem("Open");
    MenuItem close = new MenuItem("Close");
    MenuItem save = new MenuItem("Save");
    MenuItem exit = new MenuItem("Exit");

    // Add items to File menu.
    fileMenu.getItems().addAll(open, close, save,
        new SeparatorMenuItem(), exit);

    // Add keyboard accelerators for the File menu.
    open.setAccelerator(KeyCombination.keyCombination("shortcut+O"));
    close.setAccelerator(KeyCombination.keyCombination("shortcut+C"));
    save.setAccelerator(KeyCombination.keyCombination("shortcut+S"));
    exit.setAccelerator(KeyCombination.keyCombination("shortcut+E"));

    // Set action event handlers.
    open.setOnAction(MEHandler);
    close.setOnAction(MEHandler);
    save.setOnAction(MEHandler);
    exit.setOnAction(MEHandler);

    // Add File menu to the menu bar.
    mb.getMenus().add(fileMenu);
}
```

```
// Create the Options menu.
void makeOptionsMenu() {
    Menu optionsMenu = new Menu("Options");

    // Create the Colors submenu.
    Menu colorsMenu = new Menu("Colors");

    // Use check menu items for colors. This allows
    // the user to select more than one color.
    CheckMenuItem red = new CheckMenuItem("Red");
    CheckMenuItem green = new CheckMenuItem("Green");
    CheckMenuItem blue = new CheckMenuItem("Blue");

    // Add the check menu items for the Colors menu and
    // add the colors menu to the Options menu.
    colorsMenu.getItems().addAll(red, green, blue);
    optionsMenu.getItems().add(colorsMenu);

    // Select green for the default color selection.
    green.setSelected(true);

    // Create the Priority submenu.
    Menu priorityMenu = new Menu("Priority");

    // Use radio menu items for the priority setting.
    // This lets the menu show which priority is used
    // and also ensures that one and only one priority
    // can be selected at any one time.
    RadioMenuItem high = new RadioMenuItem("High");
    RadioMenuItem low = new RadioMenuItem("Low");

    // Create a toggle group and use it for the radio menu items.
    ToggleGroup tg = new ToggleGroup();
    high.setToggleGroup(tg);
    low.setToggleGroup(tg);

    // Select High priority for the default selection.
    high.setSelected(true);

    // Add the radio menu items to the Priority menu and
    // add the Priority menu to the Options menu.
    priorityMenu.getItems().addAll(high, low);
    optionsMenu.getItems().add(priorityMenu);

    // Add a separator.
    optionsMenu.getItems().add(new SeparatorMenuItem());

    // Create the Reset menu item and add it to the Options menu.
    MenuItem reset = new MenuItem("Reset");
    optionsMenu.getItems().add(reset);

    // Set action event handlers.
    red.setOnAction(MEHandler);
    green.setOnAction(MEHandler);
    blue.setOnAction(MEHandler);
```

```
high.setOnAction(MEHandler);
low.setOnAction(MEHandler);
reset.setOnAction(MEHandler);

// Use a change listener to respond to changes in the radio
// menu item setting.
tg.selectedToggleProperty().addListener(new ChangeListener<Toggle>() {
    public void changed(ObservableValue<? extends Toggle> changed,
                        Toggle oldVal, Toggle newVal) {
        if(newVal==null) return;

        // Cast newVal to RadioButton.
        RadioMenuItem rmi = (RadioMenuItem) newVal;

        // Display the selection.
        response.setText("Priority selected is " + rmi.getText());
    }
});

// Add Options menu to the menu bar.
mb.getMenus().add(optionsMenu);
}

// Create the Help menu.
void makeHelpMenu() {

    // Create an ImageView for the image.
    ImageView aboutIV = new ImageView("aboutIcon.gif");

    // Create the Help menu.
    Menu helpMenu = new Menu("Help");

    // Create the About menu item and add it to the Help menu.
    MenuItem about = new MenuItem("About", aboutIV);
    helpMenu.getItems().add(about);

    // Set action event handler.
    about.setOnAction(MEHandler);

    // Add Help menu to the menu bar.
    mb.getMenus().add(helpMenu);
}

// Create the context menu items.
void makeContextMenu() {

    // Create the edit context menu items.
    MenuItem cut = new MenuItem("Cut");
    MenuItem copy = new MenuItem("Copy");
    MenuItem paste = new MenuItem("Paste");

    // Create a context (i.e., popup) menu that shows edit options.
    editMenu = new ContextMenu(cut, copy, paste);
}
```

```

// Set the action event handlers.
cut.setOnAction(MEHandler);
copy.setOnAction(MEHandler);
paste.setOnAction(MEHandler);
}

// Create the toolbar.
void makeToolBar() {
    // Create toolbar items.
    Button btnSet = new Button("Set Breakpoint",
                               new ImageView("setBP.gif"));
    Button btnClear = new Button("Clear Breakpoint",
                                new ImageView("clearBP.gif"));
    Button btnResume = new Button("Resume Execution",
                                 new ImageView("resume.gif"));

    // Turn off text in the buttons.
    btnSet.setContentDisplay(ContentDisplay.GRAPHIC_ONLY);
    btnClear.setContentDisplay(ContentDisplay.GRAPHIC_ONLY);
    btnResume.setContentDisplay(ContentDisplay.GRAPHIC_ONLY);

    // Set tooltips.
    btnSet.setTooltip(new Tooltip("Set a breakpoint."));
    btnClear.setTooltip(new Tooltip("Clear a breakpoint."));
    btnResume.setTooltip(new Tooltip("Resume execution."));

    // Create the toolbar.
    tbDebug = new ToolBar(btnSet, btnClear, btnResume);

    // Create a handler for the toolbar buttons.
    EventHandler<ActionEvent> btnHandler = new EventHandler<ActionEvent>() {
        public void handle(ActionEvent ae) {
            response.setText(((Button)ae.getTarget()).getText());
        }
    };

    // Set the toolbar button action event handlers.
    btnSet.setOnAction(btnHandler);
    btnClear.setOnAction(btnHandler);
    btnResume.setOnAction(btnHandler);
}
}

```

Continuing Your Exploration of JavaFX

JavaFX represents a major advance in GUI frameworks for Java. It also redefines aspects of the Java platform. The preceding three chapters have introduced several of its core features, but there is much left to explore. For example, JavaFX supplies several more controls, such as sliders, stand-alone scrollbars, and tables. You will want to experiment with its layouts, such as **VBox** and **Hbox**. You will also want to explore, in detail, the various effects in **javafx.scene.effect** and the various transforms in **javafx.scene.transform**. Another exciting class is **WebView**, which gives you an easy way to integrate web content into a scene graph. Frankly, all of JavaFX is worthy of serious study. In many ways, it is charting the future course of Java.

PART

V

Applying Java

CHAPTER 37

Java Beans

CHAPTER 38

Introducing Servlets

APPENDIX

Using Java's

Documentation

Comments

This page has been intentionally left blank

CHAPTER

37

Java Beans

This chapter provides an overview of Java Beans. Beans are important because they allow you to build complex systems from software components. These components may be provided by you or supplied by one or more different vendors. Java Beans defines an architecture that specifies how these building blocks can operate together.

To better understand the value of Beans, consider the following. Hardware designers have a wide variety of components that can be integrated together to construct a system. Resistors, capacitors, and inductors are examples of simple building blocks. Integrated circuits provide more advanced functionality. All of these different parts can be reused. It is not necessary or possible to rebuild these capabilities each time a new system is needed. Also, the same pieces can be used in different types of circuits. This is possible because the behavior of these components is understood and documented.

The software industry has also been seeking the benefits of reusability and interoperability of a component-based approach. To realize these benefits, a component architecture is needed that allows programs to be assembled from software building blocks, perhaps provided by different vendors. It must also be possible for a designer to select a component, understand its capabilities, and incorporate it into an application. When a new version of a component becomes available, it should be easy to incorporate this functionality into existing code. Fortunately, Java Beans provides just such an architecture.

What Is a Java Bean?

A *Java Bean* is a software component that has been designed to be reusable in a variety of different environments. There is no restriction on the capability of a Bean. It may perform a simple function, such as obtaining an inventory value, or a complex function, such as forecasting the performance of a stock portfolio. A Bean may be visible to an end user. One example of this is a button on a graphical user interface. A Bean may also be invisible to a user. Software to decode a stream of multimedia information in real time is an example of this type of building block. Finally, a Bean may be designed to work autonomously on a user's workstation or to work in cooperation with a set of other distributed components.

Software to generate a pie chart from a set of data points is an example of a Bean that can execute locally. However, a Bean that provides real-time price information from a stock or commodities exchange would need to work in cooperation with other distributed software to obtain its data.

Advantages of Java Beans

The following list enumerates some of the benefits that Java Bean technology provides for a component developer:

- A Bean obtains all the benefits of Java's "write-once, run-anywhere" paradigm.
- The properties, events, and methods of a Bean that are exposed to another application can be controlled.
- Auxiliary software can be provided to help configure a Bean. This software is only needed when the design-time parameters for that component are being set. It does not need to be included in the run-time environment.
- The state of a Bean can be saved in persistent storage and restored at a later time.
- A Bean may register to receive events from other objects and can generate events that are sent to other objects.

Introspection

At the core of Java Beans is *introspection*. This is the process of analyzing a Bean to determine its capabilities. This is an essential feature of the Java Beans API because it allows another application, such as a design tool, to obtain information about a component. Without introspection, the Java Beans technology could not operate.

There are two ways in which the developer of a Bean can indicate which of its properties, events, and methods should be exposed. With the first method, simple naming conventions are used. These allow the introspection mechanisms to infer information about a Bean. In the second way, an additional class that extends the **BeanInfo** interface is provided that explicitly supplies this information. Both approaches are examined here.

Design Patterns for Properties

A *property* is a subset of a Bean's state. The values assigned to the properties determine the behavior and appearance of that component. A property is set through a *setter* method. A property is obtained by a *getter* method. There are two types of properties: simple and indexed.

Simple Properties

A simple property has a single value. It can be identified by the following design patterns, where N is the name of the property and T is its type:

```
public T getN( )
public void setN(T arg)
```

A read/write property has both of these methods to access its values. A read-only property has only a get method. A write-only property has only a set method.

Here are three read/write simple properties along with their getter and setter methods:

```
private double depth, height, width;

public double getDepth( ) {
    return depth;
}
public void setDepth(double d) {
    depth = d;
}

public double getHeight( ) {
    return height;
}
public void setHeight(double h) {
    height = h;
}

public double getWidth( ) {
    return width;
}
public void setWidth(double w) {
    width = w;
}
```

NOTE For a **boolean** property, a method of the form `isPropertyName()` can also be used as an accessor.

Indexed Properties

An indexed property consists of multiple values. It can be identified by the following design patterns, where **N** is the name of the property and **T** is its type:

```
public T getN(int index);
public void setN(int index, T value);
public T[ ] getN( );
public void setN(T values[ ]);
```

Here is an indexed property called **data** along with its getter and setter methods:

```
private double data[ ];

public double getData(int index) {
    return data[index];
}
public void setData(int index, double value) {
    data[index] = value;
}
public double[ ] getData( ) {
    return data;
}
public void setData(double[ ] values) {
    data = new double[values.length];
    System.arraycopy(values, 0, data, 0, values.length);
}
```

Design Patterns for Events

Beans use the delegation event model that was discussed earlier in this book. Beans can generate events and send them to other objects. These can be identified by the following design patterns, where **T** is the type of the event:

```
public void addTListener(TListener eventListener)
public void addTListener(TListener eventListener)
    throws java.util.TooManyListenersException
public void removeTListener(TListener eventListener)
```

These methods are used to add or remove a listener for the specified event. The version of **addTListener()** that does not throw an exception can be used to *multicast* an event, which means that more than one listener can register for the event notification. The version that throws **TooManyListenersException** *unicasts* the event, which means that the number of listeners can be restricted to one. In either case, **removeTListener()** is used to remove the listener. For example, assuming an event interface type called **TemperatureListener**, a Bean that monitors temperature might supply the following methods:

```
public void addTemperatureListener(TemperatureListener t1) {
    ...
}
public void removeTemperatureListener(TemperatureListener t1) {
    ...
}
```

Methods and Design Patterns

Design patterns are not used for naming nonproperty methods. The introspection mechanism finds all of the public methods of a Bean. Protected and private methods are not presented.

Using the BeanInfo Interface

As the preceding discussion shows, design patterns *implicitly* determine what information is available to the user of a Bean. The **BeanInfo** interface enables you to *explicitly* control what information is available. The **BeanInfo** interface defines several methods, including these:

```
PropertyDescriptor[ ] getPropertyDescriptors( )
EventSetDescriptor[ ] getEventSetDescriptors( )
MethodDescriptor[ ] getMethodDescriptors( )
```

They return arrays of objects that provide information about the properties, events, and methods of a Bean. The classes **PropertyDescriptor**, **EventSetDescriptor**, and **MethodDescriptor** are defined within the **java.beans** package, and they describe the indicated elements. By implementing these methods, a developer can designate exactly what is presented to a user, bypassing introspection based on design patterns.

When creating a class that implements **BeanInfo**, you must call that class *bnameBeanInfo*, where *bname* is the name of the Bean. For example, if the Bean is called **MyBean**, then the information class must be called **MyBeanBeanInfo**.

To simplify the use of **BeanInfo**, JavaBeans supplies the **SimpleBeanInfo** class. It provides default implementations of the **BeanInfo** interface, including the three methods just shown. You can extend this class and override one or more of the methods to explicitly control what aspects of a Bean are exposed. If you don't override a method, then design-pattern introspection will be used. For example, if you don't override **getPropertyDescriptors()**, then design patterns are used to discover a Bean's properties. You will see **SimpleBeanInfo** in action later in this chapter.

Bound and Constrained Properties

A Bean that has a *bound* property generates an event when the property is changed. The event is of type **PropertyChangeEvent** and is sent to objects that previously registered an interest in receiving such notifications. A class that handles this event must implement the **PropertyChangeListener** interface.

A Bean that has a *constrained* property generates an event when an attempt is made to change its value. It also generates an event of type **PropertyChangeEvent**. It too is sent to objects that previously registered an interest in receiving such notifications. However, those other objects have the ability to veto the proposed change by throwing a **PropertyVetoException**. This capability allows a Bean to operate differently according to its run-time environment. A class that handles this event must implement the **VetoableChangeListener** interface.

Persistence

Persistence is the ability to save the current state of a Bean, including the values of a Bean's properties and instance variables, to nonvolatile storage and to retrieve them at a later time. The object serialization capabilities provided by the Java class libraries are used to provide persistence for Beans.

The easiest way to serialize a Bean is to have it implement the **java.io.Serializable** interface, which is simply a marker interface. Implementing **java.io.Serializable** makes serialization automatic. Your Bean need take no other action. Automatic serialization can also be inherited. Therefore, if any superclass of a Bean implements **java.io.Serializable**, then automatic serialization is obtained.

When using automatic serialization, you can selectively prevent a field from being saved through the use of the **transient** keyword. Thus, data members of a Bean specified as **transient** will not be serialized.

If a Bean does not implement **java.io.Serializable**, you must provide serialization yourself, such as by implementing **java.io.Externalizable**. Otherwise, containers cannot save the configuration of your component.

Customizers

A Bean developer can provide a *customizer* that helps another developer configure the Bean. A customizer can provide a step-by-step guide through the process that must be followed to use the component in a specific context. Online documentation can also be provided. A Bean developer has great flexibility to develop a customizer that can differentiate his or her product in the marketplace.

Interface	Description
AppletInitializer	Methods in this interface are used to initialize Beans that are also applets.
BeanInfo	This interface allows a designer to specify information about the properties, events, and methods of a Bean.
Customizer	This interface allows a designer to provide a graphical user interface through which a Bean may be configured.
DesignMode	Methods in this interface determine if a Bean is executing in design mode.
ExceptionListener	A method in this interface is invoked when an exception has occurred.
PropertyChangeListener	A method in this interface is invoked when a bound property is changed.
PropertyEditor	Objects that implement this interface allow designers to change and display property values.
VetoableChangeListener	A method in this interface is invoked when a constrained property is changed.
Visibility	Methods in this interface allow a Bean to execute in environments where a graphical user interface is not available.

Table 37-1 The Interfaces in `java.beans`

The Java Beans API

The Java Beans functionality is provided by a set of classes and interfaces in the `java.beans` package. This section provides a brief overview of its contents. Table 37-1 lists the interfaces in `java.beans` and provides a brief description of their functionality. Table 37-2 lists the classes in `java.beans`.

Class	Description
BeanDescriptor	This class provides information about a Bean. It also allows you to associate a customizer with a Bean.
Beans	This class is used to obtain information about a Bean.
DefaultPersistenceDelegate	A concrete subclass of PersistenceDelegate .
Encoder	Encodes the state of a set of Beans. Can be used to write this information to a stream.
EventHandler	Supports dynamic event listener creation.
EventSetDescriptor	Instances of this class describe an event that can be generated by a Bean.
Expression	Encapsulates a call to a method that returns a result.
FeatureDescriptor	This is the superclass of the PropertyDescriptor , EventSetDescriptor , and MethodDescriptor classes, among others.

Table 37-2 The Classes in `java.beans`

Class	Description
IndexedPropertyChangeEvent	A subclass of PropertyChangeEvent that represents a change to an indexed property.
IndexedPropertyDescriptor	Instances of this class describe an indexed property of a Bean.
IntrospectionException	An exception of this type is generated if a problem occurs when analyzing a Bean.
Introspector	This class analyzes a Bean and constructs a BeanInfo object that describes the component.
MethodDescriptor	Instances of this class describe a method of a Bean.
ParameterDescriptor	Instances of this class describe a method parameter.
PersistenceDelegate	Handles the state information of an object.
PropertyChangeEvent	This event is generated when bound or constrained properties are changed. It is sent to objects that registered an interest in these events and that implement either the PropertyChangeListener or VetoableChangeListener interfaces.
PropertyChangeListenerProxy	Extends EventListenerProxy and implements PropertyChangeListener .
PropertyChangeSupport	Beans that support bound properties can use this class to notify PropertyChangeListener objects.
PropertyDescriptor	Instances of this class describe a property of a Bean.
PropertyEditorManager	This class locates a PropertyEditor object for a given type.
PropertyEditorSupport	This class provides functionality that can be used when writing property editors.
PropertyVetoException	An exception of this type is generated if a change to a constrained property is vetoed.
SimpleBeanInfo	This class provides functionality that can be used when writing BeanInfo classes.
Statement	Encapsulates a call to a method.
VetoableChangeListenerProxy	Extends EventListenerProxy and implements VetoableChangeListener .
VetoableChangeSupport	Beans that support constrained properties can use this class to notify VetoableChangeListener objects.
XMLDecoder	Used to read a Bean from an XML document.
XMLEncoder	Used to write a Bean to an XML document.

Table 37-2 The Classes in **java.beans** (*continued*)

Although it is beyond the scope of this chapter to discuss all of the classes, four are of particular interest: **Introspector**, **PropertyDescriptor**, **EventSetDescriptor**, and **MethodDescriptor**. Each is briefly examined here.

Introspector

The **Introspector** class provides several static methods that support introspection. Of most interest is **getBeanInfo()**. This method returns a **BeanInfo** object that can be used to obtain information about the Bean. The **getBeanInfo()** method has several forms, including the one shown here:

```
static BeanInfo getBeanInfo(Class<?> bean) throws IntrospectionException
```

The returned object contains information about the Bean specified by *bean*.

PropertyDescriptor

The **PropertyDescriptor** class describes the characteristics of a Bean property. It supports several methods that manage and describe properties. For example, you can determine if a property is bound by calling **isBound()**. To determine if a property is constrained, call **isConstrained()**. You can obtain the name of a property by calling **getName()**.

EventSetDescriptor

The **EventSetDescriptor** class represents a Bean event. It supports several methods that obtain the methods that a Bean uses to add or remove event listeners, and to otherwise manage events. For example, to obtain the method used to add listeners, call **getAddListenerMethod()**. To obtain the method used to remove listeners, call **getRemoveListenerMethod()**. To obtain the type of a listener, call **getListenerType()**. You can obtain the name of an event by calling **getName()**.

MethodDescriptor

The **MethodDescriptor** class represents a Bean method. To obtain the name of the method, call **getName()**. You can obtain information about the method by calling **getMethod()**, shown here:

```
Method getMethod()
```

An object of type **Method** that describes the method is returned.

A Bean Example

This chapter concludes with an example that illustrates various aspects of Bean programming, including introspection and using a **BeanInfo** class. It also makes use of the **Introspector**, **PropertyDescriptor**, and **EventSetDescriptor** classes. The example uses three classes. The first is a Bean called **Colors**, shown here:

```
// A simple Bean.
import java.awt.*;
import java.awt.event.*;
import java.io.Serializable;

public class Colors extends Canvas implements Serializable {
    transient private Color color; // not persistent
    private boolean rectangular; // is persistent
```

```
public Colors() {
    addMouseListener(new MouseAdapter() {
        public void mousePressed(MouseEvent me) {
            change();
        }
    });
    rectangular = false;
    setSize(200, 100);
    change();
}

public boolean getRectangular() {
    return rectangular;
}

public void setRectangular(boolean flag) {
    this.rectangular = flag;
    repaint();
}

public void change() {
    color = randomColor();
    repaint();
}

private Color randomColor() {
    int r = (int)(255*Math.random());
    int g = (int)(255*Math.random());
    int b = (int)(255*Math.random());
    return new Color(r, g, b);
}

public void paint(Graphics g) {
    Dimension d = getSize();
    int h = d.height;
    int w = d.width;
    g.setColor(color);
    if(rectangular) {
        g.fillRect(0, 0, w-1, h-1);
    }
    else {
        g.fillOval(0, 0, w-1, h-1);
    }
}
```

The **Colors** Bean displays a colored object within a frame. The color of the component is determined by the private **Color** variable **color**, and its shape is determined by the private **boolean** variable **rectangular**. The constructor defines an anonymous inner class that extends **MouseAdapter** and overrides its **mousePressed()** method. The **change()** method is invoked in response to mouse presses. It selects a random color and then repaints the component. The **getRectangular()** and **setRectangular()** methods provide access to the one property

of this Bean. The **change()** method calls **randomColor()** to choose a color and then calls **repaint()** to make the change visible. Notice that the **paint()** method uses the **rectangular** and **color** variables to determine how to present the Bean.

The next class is **ColorsBeanInfo**. It is a subclass of **SimpleBeanInfo** that provides explicit information about **Colors**. It overrides **getPropertyDescriptors()** in order to designate which properties are presented to a Bean user. In this case, the only property exposed is **rectangular**. The method creates and returns a **PropertyDescriptor** object for the **rectangular** property. The **PropertyDescriptor** constructor that is used is shown here:

```
PropertyDescriptor(String property, Class<?> beanCls)
    throws IntrospectionException
```

Here, the first argument is the name of the property, and the second argument is the class of the Bean.

```
// A Bean information class.
import java.beans.*;
public class ColorsBeanInfo extends SimpleBeanInfo {
    public PropertyDescriptor[] getPropertyDescriptors() {
        try {
            PropertyDescriptor rectangular = new
                PropertyDescriptor("rectangular", Colors.class);
            PropertyDescriptor pd[] = {rectangular};
            return pd;
        }
        catch(Exception e) {
            System.out.println("Exception caught. " + e);
        }
        return null;
    }
}
```

The final class is called **IntrospectorDemo**. It uses introspection to display the properties and events that are available within the **Colors** Bean.

```
// Show properties and events.
import java.awt.*;
import java.beans.*;

public class IntrospectorDemo {
    public static void main(String args[]) {
        try {
            Class<?> c = Class.forName("Colors");
            BeanInfo beanInfo = Introspector.getBeanInfo(c);

            System.out.println("Properties:");
            PropertyDescriptor propertyDescriptor[] =
                beanInfo.getPropertyDescriptors();
            for(int i = 0; i < propertyDescriptor.length; i++) {
                System.out.println("\t" + propertyDescriptor[i].getName());
            }
        }
    }
}
```

```
        System.out.println("Events:");
        EventSetDescriptor eventSetDescriptor[] =
            beanInfo.getEventSetDescriptors();
        for(int i = 0; i < eventSetDescriptor.length; i++) {
            System.out.println("\t" + eventSetDescriptor[i].getName());
        }
    }
    catch(Exception e) {
        System.out.println("Exception caught. " + e);
    }
}
```

The output from this program is the following:

```
Properties:
    rectangular
Events:
    mouseWheel
    mouse
    mouseMotion
    component
    hierarchyBounds
    focus
    hierarchy
    propertyChange
    inputMethod
    key
```

Notice two things in the output. First, because **ColorsBeanInfo** overrides **getPropertyDescriptors()** such that the only property returned is **rectangular**, only the **rectangular** property is displayed. However, because **getEventSetDescriptors()** is not overridden by **ColorsBeanInfo**, design-pattern introspection is used, and all events are found, including those in **Colors**' superclass, **Canvas**. Remember, if you don't override one of the "get" methods defined by **SimpleBeanInfo**, then the default, design-pattern introspection is used. To observe the difference that **ColorsBeanInfo** makes, erase its class file and then run **IntrospectorDemo** again. This time it will report more properties.

This page has been intentionally left blank

CHAPTER

38

Introducing Servlets

This chapter presents an introduction to *Servlets*. Servlets are small programs that execute on the server side of a web connection. Just as applets dynamically extend the functionality of a web browser, servlets dynamically extend the functionality of a web server. The topic of servlets is quite large, and it is beyond the scope of this chapter to cover it all. Instead, we will focus on the core concepts, interfaces, and classes, and develop several examples.

Background

In order to understand the advantages of servlets, you must have a basic understanding of how web browsers and servers cooperate to provide content to a user. Consider a request for a static web page. A user enters a Uniform Resource Locator (URL) into a browser. The browser generates an HTTP request to the appropriate web server. The web server maps this request to a specific file. That file is returned in an HTTP response to the browser. The HTTP header in the response indicates the type of the content. The Multipurpose Internet Mail Extensions (MIME) are used for this purpose. For example, ordinary ASCII text has a MIME type of text/plain. The Hypertext Markup Language (HTML) source code of a web page has a MIME type of text/html.

Now consider dynamic content. Assume that an online store uses a database to store information about its business. This would include items for sale, prices, availability, orders, and so forth. It wishes to make this information accessible to customers via web pages. The contents of those web pages must be dynamically generated to reflect the latest information in the database.

In the early days of the Web, a server could dynamically construct a page by creating a separate process to handle each client request. The process would open connections to one or more databases in order to obtain the necessary information. It communicated with the web server via an interface known as the Common Gateway Interface (CGI). CGI allowed the separate process to read data from the HTTP request and write data to the HTTP response. A variety of different languages were used to build CGI programs. These included C, C++, and Perl.

However, CGI suffered serious performance problems. It was expensive in terms of processor and memory resources to create a separate process for each client request. It was also expensive to open and close database connections for each client request. In addition, the CGI programs were not platform-independent. Therefore, other techniques were introduced. Among these are servlets.

Servlets offer several advantages in comparison with CGI. First, performance is significantly better. Servlets execute within the address space of a web server. It is not necessary to create a separate process to handle each client request. Second, servlets are platform-independent because they are written in Java. Third, the Java security manager on the server enforces a set of restrictions to protect the resources on a server machine. Finally, the full functionality of the Java class libraries is available to a servlet. It can communicate with applets, databases, or other software via the sockets and RMI mechanisms that you have seen already.

The Life Cycle of a Servlet

Three methods are central to the life cycle of a servlet. These are **init()**, **service()**, and **destroy()**. They are implemented by every servlet and are invoked at specific times by the server. Let us consider a typical user scenario to understand when these methods are called.

First, assume that a user enters a Uniform Resource Locator (URL) to a web browser. The browser then generates an HTTP request for this URL. This request is then sent to the appropriate server.

Second, this HTTP request is received by the web server. The server maps this request to a particular servlet. The servlet is dynamically retrieved and loaded into the address space of the server.

Third, the server invokes the **init()** method of the servlet. This method is invoked only when the servlet is first loaded into memory. It is possible to pass initialization parameters to the servlet so it may configure itself.

Fourth, the server invokes the **service()** method of the servlet. This method is called to process the HTTP request. You will see that it is possible for the servlet to read data that has been provided in the HTTP request. It may also formulate an HTTP response for the client.

The servlet remains in the server's address space and is available to process any other HTTP requests received from clients. The **service()** method is called for each HTTP request.

Finally, the server may decide to unload the servlet from its memory. The algorithms by which this determination is made are specific to each server. The server calls the **destroy()** method to relinquish any resources such as file handles that are allocated for the servlet. Important data may be saved to a persistent store. The memory allocated for the servlet and its objects can then be garbage collected.

Servlet Development Options

To create servlets, you will need access to a servlet container/server. Two popular ones are Glassfish and Tomcat. Glassfish is from Oracle and is provided by the Java EE SDK. It is supported by NetBeans. Tomcat is an open-source product maintained by the Apache Software Foundation. It can also be used by NetBeans. Both Tomcat and Glassfish can also be used with other IDEs, such as Eclipse. The examples and descriptions in this chapter use Tomcat for reasons that will soon be apparent.

Although IDEs such as NetBeans and Eclipse are very useful and can streamline the creation of servlets, they are not used in this chapter. The way you develop and deploy servlets differs among IDEs, and it is simply not possible for this book to address each environment. Furthermore, many readers will be using the command-line tools rather than an IDE. Therefore, if you are using an IDE, you must refer to the instructions for that environment for information concerning the development and deployment of servlets. For this reason, the instructions given here and elsewhere in this chapter assume that only the command-line tools are employed. Thus, they will work for nearly any reader.

Tomcat is used in this chapter because, in the opinion of this author, it makes it relatively easy to run the example servlets using only command-line tools and a text editor. It is also widely available in various programming environments. Furthermore, since only command-line tools are used, you don't need to download and install an IDE just to experiment with servlets. Understand, however, that even if you are developing in an environment that uses Glassfish, the concepts presented here still apply. It is just that the mechanics of preparing a servlet for testing will be slightly different.

REMEMBER The instructions for developing and deploying servlets in this chapter are based on Tomcat and use only command-line tools. If you are using an IDE and different servlet container/server, consult the documentation for your environment.

Using Tomcat

Tomcat contains the class libraries, documentation, and run-time support that you will need to create and test servlets. At the time of this writing, several versions of Tomcat are available. The instructions that follow use 7.0.47. You can download Tomcat from tomcat.apache.org. You should choose a version appropriate to your environment.

The examples in this chapter assume a 64-bit Windows environment. Assuming that a 64-bit version of Tomcat 7.0.47 was unpacked from the root directly, the default location is

```
C:\apache-tomcat-7.0.47-windows-x64\apache-tomcat-7.0.47\
```

This is the location assumed by the examples in this book. If you load Tomcat in a different location (or use a different version of Tomcat), you will need to make appropriate changes to the examples. You may need to set the environmental variable **JAVA_HOME** to the top-level directory in which the Java Development Kit is installed.

NOTE All of the directories shown in this section assume Tomcat 7.0.47. If you install a different version of Tomcat, then you will need to adjust the directory names and paths to match those used by the version you installed.

Once installed, you start Tomcat by selecting **startup.bat** from the **bin** directly under the **apache-tomcat-7.0.47** directory. To stop Tomcat, execute **shutdown.bat**, also in the **bin** directory.

The classes and interfaces needed to build servlets are contained in **servlet-api.jar**, which is in the following directory:

```
C:\apache-tomcat-7.0.47-windows-x64\apache-tomcat-7.0.47\lib
```

To make **servlet-api.jar** accessible, update your **CLASSPATH** environment variable so that it includes

```
C:\apache-tomcat-7.0.47-windows-x64\apache-tomcat-7.0.47\lib\servlet-api.jar
```

Alternatively, you can specify this file when you compile the servlets. For example, the following command compiles the first servlet example:

```
javac HelloServlet.java -classpath "C:\apache-tomcat-7.0.47-windows-x64\apache-tomcat-7.0.47\lib\servlet-api.jar"
```

Once you have compiled a servlet, you must enable Tomcat to find it. For our purposes, this means putting it into a directory under Tomcat's **webapps** directory and entering its name into a **web.xml** file. To keep things simple, the examples in this chapter use the directory and **web.xml** file that Tomcat supplies for its own example servlets. This way, you won't have to create any files or directories just to experiment with the sample servlets. Here is the procedure that you will follow.

First, copy the servlet's class file into the following directory:

```
C:\apache-tomcat-7.0.47-windows-x64\apache-tomcat-7.0.47\webapps\examples\WEB-INF\classes
```

Next, add the servlet's name and mapping to the **web.xml** file in the following directory:

```
C:\apache-tomcat-7.0.47-windows-x64\apache-tomcat-7.0.47\webapps\examples\WEB-INF
```

For instance, assuming the first example, called **HelloServlet**, you will add the following lines in the section that defines the servlets:

```
<servlet>
  <servlet-name>HelloServlet</servlet-name>
  <servlet-class>HelloServlet</servlet-class>
</servlet>
```

Next, you will add the following lines to the section that defines the servlet mappings:

```
<servlet-mapping>
  <servlet-name>HelloServlet</servlet-name>
  <url-pattern>/servlet/HelloServlet</url-pattern>
</servlet-mapping>
```

Follow this same general procedure for all of the examples.

A Simple Servlet

To become familiar with the key servlet concepts, we will begin by building and testing a simple servlet. The basic steps are the following:

1. Create and compile the servlet source code. Then, copy the servlet's class file to the proper directory, and add the servlet's name and mappings to the proper **web.xml** file.

2. Start Tomcat.
3. Start a web browser and request the servlet.

Let us examine each of these steps in detail.

Create and Compile the Servlet Source Code

To begin, create a file named **HelloServlet.java** that contains the following program:

```
import java.io.*;
import javax.servlet.*;

public class HelloServlet extends GenericServlet {

    public void service(ServletRequest request,
                        ServletResponse response)
        throws ServletException, IOException {
        response.setContentType("text/html");
        PrintWriter pw = response.getWriter();
        pw.println("<B>Hello! ");
        pw.close();
    }
}
```

Let's look closely at this program. First, note that it imports the **javax.servlet** package. This package contains the classes and interfaces required to build servlets. You will learn more about these later in this chapter. Next, the program defines **HelloServlet** as a subclass of **GenericServlet**. The **GenericServlet** class provides functionality that simplifies the creation of a servlet. For example, it provides versions of **init()** and **destroy()**, which may be used as is. You need supply only the **service()** method.

Inside **HelloServlet**, the **service()** method (which is inherited from **GenericServlet**) is overridden. This method handles requests from a client. Notice that the first argument is a **ServletRequest** object. This enables the servlet to read data that is provided via the client request. The second argument is a **ServletResponse** object. This enables the servlet to formulate a response for the client.

The call to **setContentType()** establishes the MIME type of the HTTP response. In this program, the MIME type is `text/html`. This indicates that the browser should interpret the content as HTML source code.

Next, the **getWriter()** method obtains a **PrintWriter**. Anything written to this stream is sent to the client as part of the HTTP response. Then **println()** is used to write some simple HTML source code as the HTTP response.

Compile this source code and place the **HelloServlet.class** file in the proper Tomcat directory as described in the previous section. Also, add **HelloServlet** to the **web.xml** file, as described earlier.

Start Tomcat

Start Tomcat as explained earlier. Tomcat must be running before you try to execute a servlet.

Start a Web Browser and Request the Servlet

Start a web browser and enter the URL shown here:

`http://localhost:8080/examples/servlets/servlet/HelloServlet`

Alternatively, you may enter the URL shown here:

`http://127.0.0.1:8080/examples/servlets/servlet/HelloServlet`

This can be done because 127.0.0.1 is defined as the IP address of the local machine.

You will observe the output of the servlet in the browser display area. It will contain the string **Hello!** in bold type.

The Servlet API

Two packages contain the classes and interfaces that are required to build the servlets described in this chapter. These are **javax.servlet** and **javax.servlet.http**. They constitute the core of the Servlet API. Keep in mind that these packages are not part of the Java core packages. Therefore, they are not included with Java SE. Instead, they are provided by Tomcat. They are also provided by Java EE.

The Servlet API has been in a process of ongoing development and enhancement. The current servlet specification is version 3.1. However, because changes happen fast in the world of Java, you will want to check for any additions or alterations. This chapter discusses the core of the Servlet API, which will be available to most readers and works with all modern versions of the servlet specification.

The **javax.servlet** Package

The **javax.servlet** package contains a number of interfaces and classes that establish the framework in which servlets operate. The following table summarizes several key interfaces that are provided in this package. The most significant of these is **Servlet**. All servlets must implement this interface or extend a class that implements the interface. The **ServletRequest** and **ServletResponse** interfaces are also very important.

Interface	Description
Servlet	Declares life cycle methods for a servlet.
ServletConfig	Allows servlets to get initialization parameters.
ServletContext	Enables servlets to log events and access information about their environment.
ServletRequest	Used to read data from a client request.
ServletResponse	Used to write data to a client response.

The following table summarizes the core classes that are provided in the `javax.servlet` package:

Class	Description
GenericServlet	Implements the <code>Servlet</code> and <code>ServletConfig</code> interfaces.
ServletInputStream	Encapsulates an input stream for reading requests from a client.
ServletOutputStream	Encapsulates an output stream for writing responses to a client.
ServletException	Indicates a servlet error occurred.
UnavailableException	Indicates a servlet is unavailable.

Let us examine these interfaces and classes in more detail.

The `Servlet` Interface

All servlets must implement the `Servlet` interface. It declares the `init()`, `service()`, and `destroy()` methods that are called by the server during the life cycle of a servlet. A method is also provided that allows a servlet to obtain any initialization parameters. The methods defined by `Servlet` are shown in Table 38-1.

The `init()`, `service()`, and `destroy()` methods are the life cycle methods of the servlet. These are invoked by the server. The `getServletConfig()` method is called by the servlet to obtain initialization parameters. A servlet developer overrides the `getServletInfo()` method to provide a string with useful information (for example, the version number). This method is also invoked by the server.

Method	Description
<code>void destroy()</code>	Called when the servlet is unloaded.
<code>ServletConfig getServletConfig()</code>	Returns a <code>ServletConfig</code> object that contains any initialization parameters.
<code>String getServletInfo()</code>	Returns a string describing the servlet.
<code>void init(ServletConfig sc)</code> <code>throws ServletException</code>	Called when the servlet is initialized. Initialization parameters for the servlet can be obtained from <code>sc</code> . A <code>ServletException</code> should be thrown if the servlet cannot be initialized.
<code>void service(ServletRequest req,</code> <code> ServletResponse res)</code> <code>throws ServletException,</code> <code> IOException</code>	Called to process a request from a client. The request from the client can be read from <code>req</code> . The response to the client can be written to <code>res</code> . An exception is generated if a servlet or IO problem occurs.

Table 38-1 The Methods Defined by `Servlet`

The ServletConfig Interface

The **ServletConfig** interface allows a servlet to obtain configuration data when it is loaded. The methods declared by this interface are summarized here:

Method	Description
ServletContext getServletContext()	Returns the context for this servlet.
String getInitParameter(String <i>param</i>)	Returns the value of the initialization parameter named <i>param</i> .
Enumeration<String> getInitParameterNames()	Returns an enumeration of all initialization parameter names.
String getServletName()	Returns the name of the invoking servlet.

The ServletContext Interface

The **ServletContext** interface enables servlets to obtain information about their environment. Several of its methods are summarized in Table 38-2.

The ServletRequest Interface

The **ServletRequest** interface enables a servlet to obtain information about a client request. Several of its methods are summarized in Table 38-3.

The ServletResponse Interface

The **ServletResponse** interface enables a servlet to formulate a response for a client. Several of its methods are summarized in Table 38-4.

Method	Description
Object getAttribute(String <i>attr</i>)	Returns the value of the server attribute named <i>attr</i> .
String getMimeType(String <i>file</i>)	Returns the MIME type of <i>file</i> .
String getRealPath(String <i>vpath</i>)	Returns the real (i.e., absolute) path that corresponds to the relative path <i>vpath</i> .
String getServerInfo()	Returns information about the server.
void log(String <i>s</i>)	Writes <i>s</i> to the servlet log.
void log(String <i>s</i> , Throwable <i>e</i>)	Writes <i>s</i> and the stack trace for <i>e</i> to the servlet log.
void setAttribute(String <i>attr</i> , Object <i>val</i>)	Sets the attribute specified by <i>attr</i> to the value passed in <i>val</i> .

Table 38-2 Various Methods Defined by **ServletContext**

Method	Description
Object getAttribute(String <i>attr</i>)	Returns the value of the attribute named <i>attr</i> .
String getCharacterEncoding()	Returns the character encoding of the request.
int getContentLength()	Returns the size of the request. The value -1 is returned if the size is unavailable.
String getContentType()	Returns the type of the request. A null value is returned if the type cannot be determined.
ServletInputStream getInputStream() throws IOException	Returns a ServletInputStream that can be used to read binary data from the request. An IllegalStateException is thrown if getReader() has been previously invoked on this object.
String getParameter(String <i>pname</i>)	Returns the value of the parameter named <i>pname</i> .
Enumeration<String> getParameterNames()	Returns an enumeration of the parameter names for this request.
String[] getParameterValues(String <i>name</i>)	Returns an array containing values associated with the parameter specified by <i>name</i> .
String getProtocol()	Returns a description of the protocol.
BufferedReader getReader() throws IOException	Returns a buffered reader that can be used to read text from the request. An IllegalStateException is thrown if getInputStream() has been previously invoked on this object.
String getRemoteAddr()	Returns the string equivalent of the client IP address.
String getRemoteHost()	Returns the string equivalent of the client host name.
String getScheme()	Returns the transmission scheme of the URL used for the request (for example, "http", "ftp").
String getServerName()	Returns the name of the server.
int getServerPort()	Returns the port number.

Table 38-3 Various Methods Defined by **ServletRequest**

Method	Description
String getCharacterEncoding()	Returns the character encoding for the response.
ServletOutputStream getOutputStream() throws IOException	Returns a ServletOutputStream that can be used to write binary data to the response. An IllegalStateException is thrown if getWriter() has been previously invoked on this object.
PrintWriter getWriter() throws IOException	Returns a PrintWriter that can be used to write character data to the response. An IllegalStateException is thrown if getOutputStream() has been previously invoked on this object.
void setContentLength(int <i>size</i>)	Sets the content length for the response to <i>size</i> .
void setContentType(String <i>type</i>)	Sets the content type for the response to <i>type</i> .

Table 38-4 Various Methods Defined by **ServletResponse**

The GenericServlet Class

The **GenericServlet** class provides implementations of the basic life cycle methods for a servlet. **GenericServlet** implements the **Servlet** and **ServletConfig** interfaces. In addition, a method to append a string to the server log file is available. The signatures of this method are shown here:

```
void log(String s)
void log(String s, Throwable e)
```

Here, *s* is the string to be appended to the log, and *e* is an exception that occurred.

The ServletInputStream Class

The **ServletInputStream** class extends **InputStream**. It is implemented by the servlet container and provides an input stream that a servlet developer can use to read the data from a client request. In addition to the input methods inherited from **InputStream**, a method is provided to read bytes from the stream. It is shown here:

```
int readLine(byte[ ] buffer, int offset, int size) throws IOException
```

Here, *buffer* is the array into which *size* bytes are placed starting at *offset*. The method returns the actual number of bytes read or *-1* if an end-of-stream condition is encountered.

The ServletOutputStream Class

The **ServletOutputStream** class extends **OutputStream**. It is implemented by the servlet container and provides an output stream that a servlet developer can use to write data to a client response. In addition to the output methods provided by **OutputStream**, it also defines the **print()** and **println()** methods, which output data to the stream.

The Servlet Exception Classes

javax.servlet defines two exceptions. The first is **ServletException**, which indicates that a servlet problem has occurred. The second is **UnavailableException**, which extends **ServletException**. It indicates that a servlet is unavailable.

Reading Servlet Parameters

The **ServletRequest** interface includes methods that allow you to read the names and values of parameters that are included in a client request. We will develop a servlet that illustrates their use. The example contains two files. A web page is defined in **PostParameters.html**, and a servlet is defined in **PostParametersServlet.java**.

The HTML source code for **PostParameters.html** is shown in the following listing. It defines a table that contains two labels and two text fields. One of the labels is Employee and the other is Phone. There is also a submit button. Notice that the action parameter of the form tag specifies a URL. The URL identifies the servlet to process the HTTP POST request.

```
<html>
<body>
```

```
<center>
<form name="Form1"
      method="post"
      action="http://localhost:8080/examples/servlets/
              servlet/PostParametersServlet">
<table>
<tr>
  <td><B>Employee</td>
  <td><input type= textbox name="e" size="25" value=""></td>
</tr>
<tr>
  <td><B>Phone</td>
  <td><input type= textbox name="p" size="25" value=""></td>
</tr>
</table>
<input type=submit value="Submit">
</body>
</html>
```

The source code for **PostParametersServlet.java** is shown in the following listing. The **service()** method is overridden to process client requests. The **getParameterNames()** method returns an enumeration of the parameter names. These are processed in a loop. You can see that the parameter name and value are output to the client. The parameter value is obtained via the **getParameter()** method.

```
import java.io.*;
import java.util.*;
import javax.servlet.*;

public class PostParametersServlet
extends GenericServlet {

    public void service(ServletRequest request,
                        ServletResponse response)
        throws ServletException, IOException {

        // Get print writer.
        PrintWriter pw = response.getWriter();

        // Get enumeration of parameter names.
        Enumeration e = request.getParameterNames();

        // Display parameter names and values.
        while(e.hasMoreElements()) {
            String pname = (String)e.nextElement();
            pw.print(pname + " = ");
            String pvalue = request.getParameter(pname);
            pw.println(pvalue);
        }
        pw.close();
    }
}
```

Compile the servlet. Next, copy it to the appropriate directory, and update the `web.xml` file, as previously described. Then, perform these steps to test this example:

1. Start Tomcat (if it is not already running).
2. Display the web page in a browser.
3. Enter an employee name and phone number in the text fields.
4. Submit the web page.

After following these steps, the browser will display a response that is dynamically generated by the servlet.

The `javax.servlet.http` Package

The preceding examples have used the classes and interfaces defined in `javax.servlet`, such as `ServletRequest`, `ServletResponse`, and `GenericServlet`, to illustrate the basic functionality of servlets. However, when working with HTTP, you will normally use the interfaces and classes in `javax.servlet.http`. As you will see, its functionality makes it easy to build servlets that work with HTTP requests and responses.

The following table summarizes the interfaces used in this chapter:

Interface	Description
<code>HttpServletRequest</code>	Enables servlets to read data from an HTTP request.
<code>HttpServletResponse</code>	Enables servlets to write data to an HTTP response.
<code>HttpSession</code>	Allows session data to be read and written.

The following table summarizes the classes used in this chapter. The most important of these is `HttpServlet`. Servlet developers typically extend this class in order to process HTTP requests.

Class	Description
<code>Cookie</code>	Allows state information to be stored on a client machine.
<code>HttpServlet</code>	Provides methods to handle HTTP requests and responses.

The `HttpServletRequest` Interface

The `HttpServletRequest` interface enables a servlet to obtain information about a client request. Several of its methods are shown in Table 38-5.

The `HttpServletResponse` Interface

The `HttpServletResponse` interface enables a servlet to formulate an HTTP response to a client. Several constants are defined. These correspond to the different status codes that can be assigned to an HTTP response. For example, `SC_OK` indicates that the HTTP

Method	Description
<code>String getAuthType()</code>	Returns authentication scheme.
<code>Cookie[] getCookies()</code>	Returns an array of the cookies in this request.
<code>long getDateHeader(String field)</code>	Returns the value of the date header field named <i>field</i> .
<code>String getHeader(String field)</code>	Returns the value of the header field named <i>field</i> .
<code>Enumeration<String> getHeaderNames()</code>	Returns an enumeration of the header names.
<code>int getIntHeader(String field)</code>	Returns the <code>int</code> equivalent of the header field named <i>field</i> .
<code>String getMethod()</code>	Returns the HTTP method for this request.
<code>String getPathInfo()</code>	Returns any path information that is located after the servlet path and before a query string of the URL.
<code>String getPathTranslated()</code>	Returns any path information that is located after the servlet path and before a query string of the URL after translating it to a real path.
<code>String getQueryString()</code>	Returns any query string in the URL.
<code>String getRemoteUser()</code>	Returns the name of the user who issued this request.
<code>String getRequestedSessionId()</code>	Returns the ID of the session.
<code>String getRequestURI()</code>	Returns the URI.
<code>StringBuffer getRequestURL()</code>	Returns the URL.
<code>String getServletPath()</code>	Returns that part of the URL that identifies the servlet.
<code>HttpSession getSession()</code>	Returns the session for this request. If a session does not exist, one is created and then returned.
<code>HttpSession getSession(boolean new)</code>	If <i>new</i> is <code>true</code> and no session exists, creates and returns a session for this request. Otherwise, returns the existing session for this request.
<code>boolean isRequestedSessionIdFromCookie()</code>	Returns <code>true</code> if a cookie contains the session ID. Otherwise, returns <code>false</code> .
<code>boolean isRequestedSessionIdFromURL()</code>	Returns <code>true</code> if the URL contains the session ID. Otherwise, returns <code>false</code> .
<code>boolean isRequestedSessionIdValid()</code>	Returns <code>true</code> if the requested session ID is valid in the current session context.

Table 38-5 Various Methods Defined by `HttpServletRequest`

request succeeded, and **SC_NOT_FOUND** indicates that the requested resource is not available. Several methods of this interface are summarized in Table 38-6.

The HttpSession Interface

The **HttpSession** interface enables a servlet to read and write the state information that is associated with an HTTP session. Several of its methods are summarized in Table 38-7. All of these methods throw an **IllegalStateException** if the session has already been invalidated.

Method	Description
void addCookie(Cookie <i>cookie</i>)	Adds <i>cookie</i> to the HTTP response.
boolean containsHeader(String <i>field</i>)	Returns true if the HTTP response header contains a field named <i>field</i> .
String encodeURL(String <i>url</i>)	Determines if the session ID must be encoded in the URL identified as <i>url</i> . If so, returns the modified version of <i>url</i> . Otherwise, returns <i>url</i> . All URLs generated by a servlet should be processed by this method.
String encodeRedirectURL(String <i>url</i>)	Determines if the session ID must be encoded in the URL identified as <i>url</i> . If so, returns the modified version of <i>url</i> . Otherwise, returns <i>url</i> . All URLs passed to sendRedirect() should be processed by this method.
void sendError(int <i>c</i> throws IOException)	Sends the error code <i>c</i> to the client.
void sendError(int <i>c</i> , String <i>s</i> throws IOException)	Sends the error code <i>c</i> and message <i>s</i> to the client.
void sendRedirect(String <i>url</i>) throws IOException	Redirects the client to <i>url</i> .
void setDateHeader(String <i>field</i> , long <i>msec</i>)	Adds <i>field</i> to the header with date value equal to <i>msec</i> (milliseconds since midnight, January 1, 1970, GMT).
void setHeader(String <i>field</i> , String <i>value</i>)	Adds <i>field</i> to the header with value equal to <i>value</i> .
void setIntHeader(String <i>field</i> , int <i>value</i>)	Adds <i>field</i> to the header with value equal to <i>value</i> .
void setStatus(int <i>code</i>)	Sets the status code for this response to <i>code</i> .

Table 38-6 Various Methods Defined by **HttpServletResponse**

The Cookie Class

The **Cookie** class encapsulates a cookie. A *cookie* is stored on a client and contains state information. Cookies are valuable for tracking user activities. For example, assume that a user visits an online store. A cookie can save the user's name, address, and other information. The user does not need to enter this data each time he or she visits the store.

A servlet can write a cookie to a user's machine via the **addCookie()** method of the **HttpServletResponse** interface. The data for that cookie is then included in the header of the HTTP response that is sent to the browser.

The names and values of cookies are stored on the user's machine. Some of the information that can be saved for each cookie includes the following:

- The name of the cookie
- The value of the cookie
- The expiration date of the cookie
- The domain and path of the cookie

Method	Description
Object getAttribute(String <i>attr</i>)	Returns the value associated with the name passed in <i>attr</i> . Returns null if <i>attr</i> is not found.
Enumeration<String> getAttributeNames()	Returns an enumeration of the attribute names associated with the session.
long getCreationTime()	Returns the creation time (in milliseconds since midnight, January 1, 1970, GMT) of the invoking session.
String getId()	Returns the session ID.
long getLastAccessedTime()	Returns the time (in milliseconds since midnight, January 1, 1970, GMT) when the client last made a request on the invoking session.
void invalidate()	Invalidates this session and removes it from the context.
boolean isNew()	Returns true if the server created the session and it has not yet been accessed by the client.
void removeAttribute(String <i>attr</i>)	Removes the attribute specified by <i>attr</i> from the session.
void setAttribute(String <i>attr</i> , Object <i>val</i>)	Associates the value passed in <i>val</i> with the attribute name passed in <i>attr</i> .

Table 38-7 Various Methods Defined by **HttpSession**

The expiration date determines when this cookie is deleted from the user's machine. If an expiration date is not explicitly assigned to a cookie, it is deleted when the current browser session ends.

The domain and path of the cookie determine when it is included in the header of an HTTP request. If the user enters a URL whose domain and path match these values, the cookie is then supplied to the web server. Otherwise, it is not.

There is one constructor for **Cookie**. It has the signature shown here:

```
Cookie(String name, String value)
```

Here, the name and value of the cookie are supplied as arguments to the constructor. The methods of the **Cookie** class are summarized in Table 38-8.

The **HttpServlet** Class

The **HttpServlet** class extends **GenericServlet**. It is commonly used when developing servlets that receive and process HTTP requests. The methods defined by the **HttpServlet** class are summarized in Table 38-9.

Method	Description
Object clone()	Returns a copy of this object.
String getComment()	Returns the comment.
String getDomain()	Returns the domain.
int getMaxAge()	Returns the maximum age (in seconds).
String getName()	Returns the name.
String getPath()	Returns the path.
boolean getSecure()	Returns true if the cookie is secure. Otherwise, returns false .
String getValue()	Returns the value.
int getVersion()	Returns the version.
boolean isHttpOnly()	Returns true if the cookie has the HttpOnly attribute.
void setComment(String <i>c</i>)	Sets the comment to <i>c</i> .
void setDomain(String <i>d</i>)	Sets the domain to <i>d</i> .
void setHttpOnly(boolean <i>httpOnly</i>)	If <i>httpOnly</i> is true , then the HttpOnly attribute is added to the cookie. If <i>httpOnly</i> is false , the HttpOnly attribute is removed.
void setMaxAge(int <i>secs</i>)	Sets the maximum age of the cookie to <i>secs</i> . This is the number of seconds after which the cookie is deleted.
void setPath(String <i>p</i>)	Sets the path to <i>p</i> .
void setSecure(boolean <i>secure</i>)	Sets the security flag to <i>secure</i> .
void setValue(String <i>v</i>)	Sets the value to <i>v</i> .
void setVersion(int <i>v</i>)	Sets the version to <i>v</i> .

Table 38-8 The Methods Defined by **Cookie**

Method	Description
void doDelete(HttpServletRequest <i>req</i> , HttpServletResponse <i>res</i>) throws IOException, ServletException	Handles an HTTP DELETE request.
void doGet(HttpServletRequest <i>req</i> , HttpServletResponse <i>res</i>) throws IOException, ServletException	Handles an HTTP GET request.
void doHead(HttpServletRequest <i>req</i> , HttpServletResponse <i>res</i>) throws IOException, ServletException	Handles an HTTP HEAD request.
void doOptions(HttpServletRequest <i>req</i> , HttpServletResponse <i>res</i>) throws IOException, ServletException	Handles an HTTP OPTIONS request.

Table 38-9 The Methods Defined by **HttpServlet**

Method	Description
<code>void doPost(HttpServletRequest req, HttpServletResponse res) throws IOException, ServletException</code>	Handles an HTTP POST request.
<code>void doPut(HttpServletRequest req, HttpServletResponse res) throws IOException, ServletException</code>	Handles an HTTP PUT request.
<code>void doTrace(HttpServletRequest req, HttpServletResponse res) throws IOException, ServletException</code>	Handles an HTTP TRACE request.
<code>long getLastModified(HttpServletRequest req)</code>	Returns the time (in milliseconds since midnight, January 1, 1970, GMT) when the requested resource was last modified.
<code>void service(HttpServletRequest req, HttpServletResponse res) throws IOException, ServletException</code>	Called by the server when an HTTP request arrives for this servlet. The arguments provide access to the HTTP request and response, respectively.

Table 38-9 The Methods Defined by `HttpServlet` (continued)

Handling HTTP Requests and Responses

The `HttpServlet` class provides specialized methods that handle the various types of HTTP requests. A servlet developer typically overrides one of these methods. These methods are `doDelete()`, `doGet()`, `doHead()`, `doOptions()`, `doPost()`, `doPut()`, and `doTrace()`. A complete description of the different types of HTTP requests is beyond the scope of this book. However, the GET and POST requests are commonly used when handling form input. Therefore, this section presents examples of these cases.

Handling HTTP GET Requests

Here we will develop a servlet that handles an HTTP GET request. The servlet is invoked when a form on a web page is submitted. The example contains two files. A web page is defined in `ColorGet.html`, and a servlet is defined in `ColorGetServlet.java`. The HTML source code for `ColorGet.html` is shown in the following listing. It defines a form that contains a select element and a submit button. Notice that the action parameter of the form tag specifies a URL. The URL identifies a servlet to process the HTTP GET request.

```
<html>
<body>
<center>
<form name="Form1"
      action="http://localhost:8080/examples/servlets/servlet/ColorGetServlet">
<B>Color:</B>
<select name="color" size="1">
<option value="Red">Red</option>
<option value="Green">Green</option>
```

```

<option value="Blue">Blue</option>
</select>
<br><br>
<input type=submit value="Submit">
</form>
</body>
</html>

```

The source code for **ColorGetServlet.java** is shown in the following listing. The **doGet()** method is overridden to process any HTTP GET requests that are sent to this servlet. It uses the **getParameter()** method of **HttpServletRequest** to obtain the selection that was made by the user. A response is then formulated.

```

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class ColorGetServlet extends HttpServlet {

    public void doGet(HttpServletRequest request,
                      HttpServletResponse response)
        throws ServletException, IOException {

        String color = request.getParameter("color");
        response.setContentType("text/html");
        PrintWriter pw = response.getWriter();
        pw.println("<B>The selected color is: ");
        pw.println(color);
        pw.close();
    }
}

```

Compile the servlet. Next, copy it to the appropriate directory, and update the **web.xml** file, as previously described. Then, perform these steps to test this example:

1. Start Tomcat, if it is not already running.
2. Display the web page in a browser.
3. Select a color.
4. Submit the web page.

After completing these steps, the browser will display the response that is dynamically generated by the servlet.

One other point: Parameters for an HTTP GET request are included as part of the URL that is sent to the web server. Assume that the user selects the red option and submits the form. The URL sent from the browser to the server is

`http://localhost:8080/examples/servlets/servlet/ColorGetServlet?color=Red`

The characters to the right of the question mark are known as the *query string*.

Handling HTTP POST Requests

Here we will develop a servlet that handles an HTTP POST request. The servlet is invoked when a form on a web page is submitted. The example contains two files. A web page is defined in **ColorPost.html**, and a servlet is defined in **ColorPostServlet.java**.

The HTML source code for **ColorPost.html** is shown in the following listing. It is identical to **ColorGet.html** except that the method parameter for the form tag explicitly specifies that the POST method should be used, and the action parameter for the form tag specifies a different servlet.

```
<html>
<body>
<center>
<form name="Form1"
      method="post"
      action="http://localhost:8080/examples/servlets/servlet/ColorPostServlet">
<B>Color:</B>
<select name="color" size="1">
<option value="Red">Red</option>
<option value="Green">Green</option>
<option value="Blue">Blue</option>
</select>
<br><br>
<input type=submit value="Submit">
</form>
</body>
</html>
```

The source code for **ColorPostServlet.java** is shown in the following listing. The **doPost()** method is overridden to process any HTTP POST requests that are sent to this servlet. It uses the **getParameter()** method of **HttpServletRequest** to obtain the selection that was made by the user. A response is then formulated.

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class ColorPostServlet extends HttpServlet {

    public void doPost(HttpServletRequest request,
                       HttpServletResponse response)
        throws ServletException, IOException {

        String color = request.getParameter("color");
        response.setContentType("text/html");
        PrintWriter pw = response.getWriter();
        pw.println("<B>The selected color is: ");
        pw.println(color);
        pw.close();
    }
}
```

Compile the servlet and perform the same steps as described in the previous section to test it.

NOTE Parameters for an HTTP POST request are not included as part of the URL that is sent to the web server. In this example, the URL sent from the browser to the server is `http://localhost:8080/examples/servlets/servlet/ColorPostServlet`. The parameter names and values are sent in the body of the HTTP request.

Using Cookies

Now, let's develop a servlet that illustrates how to use cookies. The servlet is invoked when a form on a web page is submitted. The example contains three files as summarized here:

File	Description
AddCookie.html	Allows a user to specify a value for the cookie named MyCookie .
AddCookieServlet.java	Processes the submission of AddCookie.html .
GetCookiesServlet.java	Displays cookie values.

The HTML source code for **AddCookie.html** is shown in the following listing. This page contains a text field in which a value can be entered. There is also a submit button on the page. When this button is pressed, the value in the text field is sent to **AddCookieServlet** via an HTTP POST request.

```
<html>
<body>
<center>
<form name="Form1"
method="post"
action="http://localhost:8080/examples/servlets/servlet/AddCookieServlet">
<B>Enter a value for MyCookie:</B>
<input type= textbox name="data" size=25 value="" >
<input type=submit value="Submit">
</form>
</body>
</html>
```

The source code for **AddCookieServlet.java** is shown in the following listing. It gets the value of the parameter named "data". It then creates a **Cookie** object that has the name "MyCookie" and contains the value of the "data" parameter. The cookie is then added to the header of the HTTP response via the **addCookie()** method. A feedback message is then written to the browser.

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class AddCookieServlet extends HttpServlet {

    public void doPost(HttpServletRequest request,
                      HttpServletResponse response)
        throws ServletException, IOException {
```

```
// Get parameter from HTTP request.  
String data = request.getParameter("data");  
  
// Create cookie.  
Cookie cookie = new Cookie("MyCookie", data);  
  
// Add cookie to HTTP response.  
response.addCookie(cookie);  
  
// Write output to browser.  
response.setContentType("text/html");  
PrintWriter pw = response.getWriter();  
pw.println("<B>MyCookie has been set to");  
pw.println(data);  
pw.close();  
}  
}
```

The source code for **GetCookiesServlet.java** is shown in the following listing. It invokes the **getCookies()** method to read any cookies that are included in the HTTP GET request. The names and values of these cookies are then written to the HTTP response. Observe that the **getName()** and **getValue()** methods are called to obtain this information.

```
import java.io.*;  
import javax.servlet.*;  
import javax.servlet.http.*;  
  
public class GetCookiesServlet extends HttpServlet {  
  
    public void doGet(HttpServletRequest request,  
                      HttpServletResponse response)  
        throws ServletException, IOException {  
  
        // Get cookies from header of HTTP request.  
        Cookie[] cookies = request.getCookies();  
  
        // Display these cookies.  
        response.setContentType("text/html");  
        PrintWriter pw = response.getWriter();  
        pw.println("<B>");  
        for(int i = 0; i < cookies.length; i++) {  
            String name = cookies[i].getName();  
            String value = cookies[i].getValue();  
            pw.println("name = " + name +  
                      "; value = " + value);  
        }  
        pw.close();  
    }  
}
```

Compile the servlets. Next, copy them to the appropriate directory, and update the **web.xml** file, as previously described. Then, perform these steps to test this example:

1. Start Tomcat, if it is not already running.
2. Display **AddCookie.html** in a browser.
3. Enter a value for **MyCookie**.
4. Submit the web page.

After completing these steps, you will observe that a feedback message is displayed by the browser.

Next, request the following URL via the browser:

`http://localhost:8080/examples/servlets/servlet/GetCookiesServlet`

Observe that the name and value of the cookie are displayed in the browser.

In this example, an expiration date is not explicitly assigned to the cookie via the **setMaxAge()** method of **Cookie**. Therefore, the cookie expires when the browser session ends. You can experiment by using **setMaxAge()** and observe that the cookie is then saved on the client machine.

Session Tracking

HTTP is a stateless protocol. Each request is independent of the previous one. However, in some applications, it is necessary to save state information so that information can be collected from several interactions between a browser and a server. Sessions provide such a mechanism.

A session can be created via the **getSession()** method of **HttpServletRequest**. An **HttpSession** object is returned. This object can store a set of bindings that associate names with objects. The **setAttribute()**, **getAttribute()**, **getAttributeNames()**, and **removeAttribute()** methods of **HttpSession** manage these bindings. Session state is shared by all servlets that are associated with a client.

The following servlet illustrates how to use session state. The **getSession()** method gets the current session. A new session is created if one does not already exist. The **getAttribute()** method is called to obtain the object that is bound to the name "date". That object is a **Date** object that encapsulates the date and time when this page was last accessed. (Of course, there is no such binding when the page is first accessed.) A **Date** object encapsulating the current date and time is then created. The **setAttribute()** method is called to bind the name "date" to this object.

```
import java.io.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class DateServlet extends HttpServlet {
```

```
public void doGet(HttpServletRequest request,
                  HttpServletResponse response)
throws ServletException, IOException {

    // Get the HttpSession object.
    HttpSession hs = request.getSession(true);

    // Get writer.
    response.setContentType("text/html");
    PrintWriter pw = response.getWriter();
    pw.print("<B>");

    // Display date/time of last access.
    Date date = (Date)hs.getAttribute("date");
    if(date != null) {
        pw.print("Last access: " + date + "<br>");
    }

    // Display current date/time.
    date = new Date();
    hs.setAttribute("date", date);
    pw.println("Current date: " + date);
}

}
```

When you first request this servlet, the browser displays one line with the current date and time information. On subsequent invocations, two lines are displayed. The first line shows the date and time when the servlet was last accessed. The second line shows the current date and time.

This page has been intentionally left blank

APPENDIX

Using Java's Documentation Comments

As explained in Part I, Java supports three types of comments. The first two are the // and the /* */. The third type is called a *documentation comment*. It begins with the character sequence /**. It ends with */. Documentation comments allow you to embed information about your program into the program itself. You can then use the **javadoc** utility program (supplied with the JDK) to extract the information and put it into an HTML file.

Documentation comments make it convenient to document your programs. You have almost certainly seen documentation generated with **javadoc**, because that is the way the Java API library was documented.

The **javadoc** Tags

The **javadoc** utility recognizes the following tags:

Tag	Meaning
@author	Identifies the author.
{@code}	Displays information as-is, without processing HTML styles, in code font.
@deprecated	Specifies that a program element is deprecated.
{@docRoot}	Specifies the path to the root directory of the current documentation.
@exception	Identifies an exception thrown by a method or constructor.
{@inheritDoc}	Inherits a comment from the immediate superclass.
{@link}	Inserts an in-line link to another topic.
{@linkplain}	Inserts an in-line link to another topic, but the link is displayed in a plain-text font.
{@literal}	Displays information as-is, without processing HTML styles.
@param	Documents a parameter.
@return	Documents a method's return value.
@see	Specifies a link to another topic.

Tag	Meaning
@serial	Documents a default serializable field.
@serialData	Documents the data written by the <code>writeObject()</code> or <code>writeExternal()</code> methods.
@serialField	Documents an ObjectStreamField component.
@since	States the release when a specific change was introduced.
@throws	Same as <code>@exception</code> .
{@value}	Displays the value of a constant, which must be a static field.
@version	Specifies the version of a class.

Document tags that begin with an “at” sign (@) are called *stand-alone* tags (also called *block* tags), and they must be used on their own line. Tags that begin with a brace, such as {@code}, are called *in-line* tags, and they can be used within a larger description. You may also use other, standard HTML tags in a documentation comment. However, some tags, such as headings, should not be used because they disrupt the look of the HTML file produced by **javadoc**.

As it relates to documenting source code, you can use documentation comments to document classes, interfaces, fields, constructors, and methods. In all cases, the documentation comment must immediately precede the item being documented. Some tags, such as `@see`, `@since`, and `@deprecated`, can be used to document any element. Other tags apply only to the relevant elements. Each tag is examined next.

NOTE Documentation comments can also be used for documenting a package and preparing an overview, but the procedures differ from those used to document source code. See the **javadoc** documentation for details on these uses.

@author

The `@author` tag documents the author of a class or interface. It has the following syntax:

`@author description`

Here, *description* will usually be the name of the author. You will need to specify the `-author` option when executing **javadoc** in order for the `@author` field to be included in the HTML documentation.

{@code}

The {@code} tag enables you to embed text, such as a snippet of code, into a comment. That text is then displayed as-is in code font, without any further processing, such as HTML rendering. It has the following syntax:

`{@code code-snippet}`

@deprecated

The `@deprecated` tag specifies that a program element is deprecated. It is recommended that you include `@see` or `{@link}` tags to inform the programmer about available alternatives. The syntax is the following:

`@deprecated description`

Here, *description* is the message that describes the deprecation. The **@deprecated** tag can be used in documentation for fields, methods, constructors, classes, and interfaces.

{@docRoot}

{@docRoot} specifies the path to the root directory of the current documentation.

@exception

The **@exception** tag describes an exception to a method. It has the following syntax:

```
@exception exception-name explanation
```

Here, the fully qualified name of the exception is specified by *exception-name*, and *explanation* is a string that describes how the exception can occur. The **@exception** tag can only be used in documentation for a method or constructor.

{@inheritDoc}

This tag inherits a comment from the immediate superclass.

{@link}

The {@link} tag provides an in-line link to additional information. It has the following syntax:

```
{@link pkg.class#member text}
```

Here, *pkg.class#member* specifies the name of a class or method to which a link is added, and *text* is the string that is displayed.

{@linkplain}

Inserts an in-line link to another topic. The link is displayed in plain-text font. Otherwise, it is similar to {@link}.

{@literal}

The {@literal} tag enables you to embed text into a comment. That text is then displayed as-is, without any further processing, such as HTML rendering. It has the following syntax:

```
{@literal description}
```

Here, *description* is the text that is embedded.

@param

The **@param** tag documents a parameter. It has the following syntax:

```
@param parameter-name explanation
```

Here, *parameter-name* specifies the name of a parameter. The meaning of that parameter is described by *explanation*. The **@param** tag can be used only in documentation for a method or constructor, or a generic class or interface.

@return

The **@return** tag describes the return value of a method. It has the following syntax:

@return *explanation*

Here, *explanation* describes the type and meaning of the value returned by a method. The **@return** tag can be used only in documentation for a method.

@see

The **@see** tag provides a reference to additional information. Two commonly used forms are shown here:

@see *anchor*

@see *pkg.class#member text*

In the first form, *anchor* is a link to an absolute or relative URL. In the second form, *pkg.class#member* specifies the name of the item, and *text* is the text displayed for that item. The *text* parameter is optional, and if not used, then the item specified by *pkg.class#member* is displayed. The member name, too, is optional. Thus, you can specify a reference to a package, class, or interface in addition to a reference to a specific method or field. The name can be fully qualified or partially qualified. However, the dot that precedes the member name (if it exists) must be replaced by a hash character.

@serial

The **@serial** tag defines the comment for a default serializable field. It has the following syntax:

@serial *description*

Here, *description* is the comment for that field.

@serialData

The **@serialData** tag documents the data written by the **writeObject()** and **writeExternal()** methods. It has the following syntax:

@serialData *description*

Here, *description* is the comment for that data.

@serialField

For a class that implements **Serializable**, the **@serialField** tag provides comments for an **ObjectStreamField** component. It has the following syntax:

@serialField *name type description*

Here, *name* is the name of the field, *type* is its type, and *description* is the comment for that field.

@since

The **@since** tag states that an element was introduced in a specific release. It has the following syntax:

@since *release*

Here, *release* is a string that designates the release or version in which this feature became available.

@throws

The **@throws** tag has the same meaning as the **@exception** tag.

{@value}

{@value} has two forms. The first displays the value of the constant that it precedes, which must be a **static** field. It has this form:

{@value}

The second form displays the value of a specified **static** field. It has this form:

{@value *pkg.class#field*}

Here, *pkg.class#field* specifies the name of the **static** field.

@version

The **@version** tag specifies the version of a class or interface. It has the following syntax:

@version *info*

Here, *info* is a string that contains version information, typically a version number, such as 2.2. You will need to specify the **-version** option when executing **javadoc** in order for the **@version** field to be included in the HTML documentation.

The General Form of a Documentation Comment

After the beginning `/**`, the first line or lines become the main description of your class, interface, field, constructor, or method. After that, you can include one or more of the various **@** tags. Each **@** tag must start at the beginning of a new line or follow one or more asterisks (*) that are at the start of a line. Multiple tags of the same type should be grouped together. For example, if you have three **@see** tags, put them one after the other. In-line tags (those that begin with a brace) can be used within any description.

Here is an example of a documentation comment for a class:

```
/**  
 * This class draws a bar chart.  
 * @author Herbert Schildt  
 * @version 3.2  
 */
```

What javadoc Outputs

The **javadoc** program takes as input your Java program's source file and outputs several HTML files that contain the program's documentation. Information about each class will be in its own HTML file. **javadoc** will also output an index and a hierarchy tree. Other HTML files can be generated.

An Example that Uses Documentation Comments

Following is a sample program that uses documentation comments. Notice the way each comment immediately precedes the item that it describes. After being processed by **javadoc**, the documentation about the **SquareNum** class will be found in **SquareNum.html**.

```
import java.io.*;
/**
 * This class demonstrates documentation comments.
 * @author Herbert Schildt
 * @version 1.2
 */
public class SquareNum {
    /**
     * This method returns the square of num.
     * This is a multiline description. You can use
     * as many lines as you like.
     * @param num The value to be squared.
     * @return num squared.
    */
    public double square(double num) {
        return num * num;
    }

    /**
     * This method inputs a number from the user.
     * @return The value input as a double.
     * @exception IOException On input error.
     * @see IOException
    */
    public double getNumber() throws IOException {
        // create a BufferedReader using System.in
        InputStreamReader isr = new InputStreamReader(System.in);
        BufferedReader inData = new BufferedReader(isr);
        String str;

        str = inData.readLine();
        return (new Double(str)).doubleValue();
    }

    /**
     * This method demonstrates square().
     * @param args Unused.
     * @exception IOException On input error.
     * @see IOException
    */

    public static void main(String args[])
        throws IOException
    {
        SquareNum ob = new SquareNum();
        double val;
```

```
System.out.println("Enter value to be squared: ");
val = ob.getNumber();
val = ob.square(val);

        System.out.println("Squared value is " + val);
    }
}
```

This page has been intentionally left blank

Index

- &
 bitwise AND, 66, 67, 68–69
 Boolean logical AND, 75–76
 and bounded type declarations, 349
 && (short-circuit AND), 75, 76–77
*
 and glob syntax, 715–716
 multiplication operator, 27, 61–62
 regular expression quantifier, 995
 used in import statement, 194, 333
** (glob syntax), 716
@
 annotation syntax, 280
 used with tags (javadoc), 1236, 1239
|
 bitwise OR, 66, 67, 68–69
 Boolean logical OR, 75–76
|| (short-circuit OR), 75, 76–77
[], 33, 51, 52, 54, 58
 character class specification, 995, 999
^
 bitwise exclusive OR (XOR), 66, 67, 68–69
 Boolean logical exclusive OR (XOR), 75–76
: (used with a label), 105
::
 constructor reference, 33, 404, 408
 method reference, 33, 396, 402
, (comma), 33, 95, 387
 format flag, 615, 617
{ }, 24, 25, 26, 30, 33, 45, 46, 53, 56, 81, 82, 89, 93, 217,
 291, 388
 used with javadoc tags, 1236
=, 27, 44, 74, 77
== (Boolean logical operator), 75
== (relational operator), 28, 74, 264, 269
 versus equals (), 422–423
!, 75–76
!=, 74, 75
/, 61–62
/* */, 24
/** */, 33, 1235
//, 25
<, 28, 74
 argument index syntax, 618–619
<>
 diamond operator (type inference), 372–373
 and generic type parameter, 340
<?>, 282, 283
<<, 66, 69–70
- <=, 74
-, 61–62
 format flag, 615
-> lambda expression arrow operator, 16, 61, 382
-~, 30, 61, 64–65
%
 used in format conversion specifier syntax, 607
 modulus operator, 61, 63
(format flag, 615, 617
(), 25, 33, 79, 114, 123
 used in a lambda expression, 382, 383, 386
used to raise the precedence of operations, 33, 41,
 79, 418
.dot operator, 111, 117–118, 146, 170, 188, 194, 211
in import statement, 194
in multileveled package statement, 188
and nested interfaces, 200–201
regular expression wildcard character, 995, 998
separator, 33
... (variable-length argument syntax), 156, 159
+
 addition operator, 61–62
 concatenation operator, 27, 152–153, 417–418
 format flag, 615, 616
 regular expression quantifier, 995, 997–999
 unary plus, 61, 62
++ , 30, 61, 64–66
format flag, 615, 617
?
 regular expression quantifier, 995, 998–999
 wildcard argument specifier, 350, 353, 356, 370, 379
?: (ternary if-then-else operator), 75, 77–78
>, 28, 74
>>, 66, 70–72
>>>, 66, 72–73
>=, 74
; (semicolon), 26, 33, 90, 197
 used in try-with-resources statement, 317, 650
~ (bitwise unary NOT operator), 66, 67, 68–69
_ (underscore), 32, 42, 43

A

- abs(), 131–132, 478
Abstract method(s), 182–184
 and lambda expressions, 382, 383, 384, 385
abstract type modifier, 182, 185, 199–200, 383

Abstract Window Toolkit. *See AWT*
 (Abstract Window Toolkit)

AbstractAction interface, 1091, 1092

AbstractButton class, 1045, 1047, 1048, 1070, 1073, 1078, 1081

AbstractCollection class, 511, 513, 520

AbstractList class, 511, 562

AbstractMap class, 537, 538, 539, 541

AbstractQueue class, 511, 519

AbstractSequentialList class, 511, 515

AbstractSet class, 511, 516, 518, 521

accept(), 526, 636, 646, 648, 716, 742, 972, 985, 987

Access control, 141–144

- and default access, 190, 197
- example program, 191–194
- and inheritance, 142, 144, 163–164
- and packages, 142, 187, 190–194

Access modifiers, 25, 142, 190–191

acos(), 477

acquire(), 918–921

Action (Swing), 1069, 1089–1094

Action interface, 1089

ActionEvent class, 772, 773, 836, 837, 847, 872, 1032, 1043, 1045, 1052, 1179
JavaFX, 1115, 1116, 1118

ActionListener interface, 782, 783, 836, 839, 847, 872, 1032, 1045, 1051, 1074, 1089

actionPerformed(), 783, 836, 837, 839, 1032, 1033, 1045, 1051, 1052, 1074, 1078, 1089, 1091–1092

adapt(), 962

Adapter classes, 791–793

add(), 502, 503, 504, 505, 516, 523, 588, 801, 834, 839, 844, 846, 858, 863, 871, 985, 1006, 1007, 1029, 1051, 1064, 1071, 1072, 1087, 1091, 1113, 1160, 1164, 1166, 1173, 1174, 1189

addActionListener(), 1032

addAll(), 502, 503, 504, 505, 551, 985, 1114, 1118, 1160, 1173, 1174, 1179

addCookie(), 1224, 1230

addElement(), 568

addEventFilter(), 1115

addExact(), 480

addFirst(), 509, 510, 515, 985

addImage(), 892, 893

addItem(), 1061

addKeyListener(), 770

addLast(), 509, 510, 515, 516

addListener(), 1139, 1160

addMouseListener(), 788, 793–794, 1084

addMouseMotionListener(), 770, 788

Address, Internet, 728, 729–731

addSeparator(), 1072

addSuppressed(), 228

addTab(), 1053, 1054

addTListener() 1202

addTypeListener(), 770, 771

AdjustmentEvent class, 772, 773–774, 850

AdjustmentListener interface, 782, 783, 850

adjustmentValueChanged(), 783

Affine class, 1166

Algorithms, collection, 499, 550–556, 561

ALIGN, 760, 761

allMatch(), 990

allocate(), 691, 701, 702, 720–721, 723

ALT, 760, 761

anchor constraint field, 866, 867–868

AND operator

- bitwise (&), 66, 67, 68–69
- Boolean logical (&), 75–76
- and bounded type declarations (&), 349
- short-circuit (&&), 75, 76–77

AnnotatedElement interface, 286–287, 288, 298, 496

Annotation interface, 280, 286, 496

Annotation(s), 13, 14, 279–299, 496

- built-in, 290–292
- container, 297, 298
- declaration example, 280
- marker, 288–289
- member, default value for, 287–288
- obtaining all, 285–286
- reflection to obtain, using, 281–286
- repeated, 287, 297–299
- restrictions on, 299
- retention policies, 281
- single-member, 289–290
- type, 292–297

annotationType(), 280

anyMatch(), 990

Apache Software Foundation, 1212

API library, compact profiles of the, 336

API packages, table of core Java, 991–993

append(), 435, 494, 671, 854

Appendable interface, 494, 608, 665, 670, 679

appendCodePoint(), 438

appendTo(), 465

Applet, 8, 16

Applet, AWT-based, 318–321, 747–767

- architecture, 751, 756
- basics, 747–750
- colors, setting and obtaining, 754–755
- event-driven nature of the, 751, 769
- executing an, 320–321, 747–748, 751, 760
- and the Internet, 8–9, 16, 318–319, 320, 748–749
- local, 748
- and main(), 26, 110, 320, 321, 748
- outputting to console, 767
- passing parameters to an, 761–764
- request for repainting, 756–759
- and security, 748–749
- signed, 320, 748–749
- skeleton, 751–754
- and socket connections, 731
- as source and listener for events, 788
- string output to an, 319, 748, 754, 756
- and uncaught exceptions, 762
- viewer, 320–321, 747, 748, 751, 759, 760, 767, 798, 801

Applet class, 319, 747–765, 781, 788, 792, 793, 801, 886, 887, 1033, 1035

- methods, table of, 749–750

applet package, 301, 319, 747

Applet, Swing, 747, 752, 754, 1025, 1030, 1033–1035

APPLET tag, HTML, 320, 321, 748

- full syntax for, 760–761

AppletContext interface, 747, 761, 765–766

- methods, table of, 765–766

- AppletStub interface, 747, 767
- appletviewer, 320, 747, 749, 752, 1034
 - status window, using, 759–760
- Application class, 1107–1108, 1110
- Application launcher (java), 24, 188, 189
 - and main(), 25
- apply(), 636, 637, 973, 978
- applyAsDouble(), 636, 981
- ARCHIVE, 761
- AreaAveragingScaleFilter class, 899
- areFieldsSet, 588
- Argument(s), 116, 120
 - command-line, 25, 154–155
 - index, 618–619
 - passing, 136–138
 - type. *See* Type argument(s)
 - variable-length. *See* Varargs
 - wildcard. *See* Wildcard arguments
- Arithmetic operators, 61–66
- ArithmeticException, 215, 216, 226, 479
- Array class, 496
- Array(s), 25, 51–58, 147, 185
 - boundary checks, 53
 - and collections, 556
 - constructor reference for, 408
 - converting collections into, 503, 513–514
 - copying with arraycopy(), 467, 469–470
 - declaration syntax, alternative, 58
 - dynamic, 496, 511–514, 520, 562
 - and the for-each loop, 97–101
 - and generics, 377–379
 - implemented as objects, 147
 - indexes, 52
 - initializing, 53, 56–57
 - length instance variable of, 147–149
 - multidimensional, 54–58
 - one-dimensional, 51–54
 - and spliterators, 559
 - and the stream API, 969
 - of strings, 154
 - and valueOf(), 429
 - and varargs, 156
- ArrayList class, 943
- arraycopy(), 467, 469–470
- ArrayDeque class, 511, 520–521, 568
- ArrayIndexOutOfBoundsException, 219, 226, 557
- ArrayList class, 511–514, 530, 562, 563, 969
 - example using an, 524–525
 - example using a stream API stream, 969–973, 978–982, 983–984, 986–989
- Arrays class, 556–561, 969
- ArrayStoreException, 226, 557, 558
- arrive(), 930–931
- arriveAndAwaitAdvance(), 930, 931, 933, 936
- arriveAndDeregister(), 931, 933
- Arrow operator (→), 16, 61, 382
- ASCII character set, 39, 40, 43, 424
 - and strings on the Internet, 415, 420
- asin(), 477
- asList(), 556
- Assembly language, 4, 5
- assert statement, 13, 328–331
- Assertions, 328–331
- AssertionError, 328, 329
- Assignment operator
 - =, 27, 74, 77
 - arithmetic compound (*op*=), 61, 63–64
 - bitwise compound, 66, 73–74
 - Boolean logical, 75
- atan(), 477
- atan2(), 477
- Atomic operations, 946–947
- AtomicInteger class, 917, 946–947
- AtomicLong class, 917, 946
- AttributeView interface, 699
- AudioClip interface, 747, 767
- Autoboxing/unboxing, 14, 272, 274–279, 341–342
 - Boolean and Character values, 278
 - and the Collections Framework, 500, 514
 - definition of, 274
 - and error prevention, 278–279
 - and expressions, 276–277
 - and methods, 275–276
 - when to use, 279
- AutoCloseable interface, 310, 316, 495, 619, 626, 648, 650, 651, 654, 665, 667, 668, 669, 670, 679, 683, 685, 691, 701, 714, 732, 743, 966
- Automatic resource management (ARM), 214, 315–318, 495, 619, 734
- available(), 651, 652–654, 685, 686
- availableProcessors(), 958
- await(), 923–925, 927, 944
- awaitAdvance(), 936
- awaitAdvanceInterruptibly(), 936
- AWT (Abstract Window Toolkit), 301, 319, 747, 748, 797–798, 833, 1105
 - and applet architectural constraints, 756
 - classes, table of some, 798–800
 - color system, 815
 - controls. *See* Controls, AWT
 - creating stand-alone windows with, 809–810
 - and fonts, 819–825
 - layout managers. *See* Layout manager(s)
 - support for imaging, 885
 - support for text and graphics, 811
 - and Swing, 797, 1021–1022
- AWTEvent class, 772, 798

B

- B, 4
- Base64 class, 635
- BaseStream interface, 966–968, 975
 - methods, table of, 966
- BASIC, 4
- Basic multilingual plane (BMP), 458
- BasicFileAttributes class, 698–699, 712
 - methods, table of, 698
- BasicFileAttributeView interface, 699
- BCP 47, 595
- BCPL, 4
- BeanInfo interface, 1200, 1202–1203, 1204
- Beans, Java. *See* Java Beans
- Bell curve, 596
- Bell Laboratories, 6
- Berkeley UNIX, 727

- Berners-Lee, Tim, 735
Beyond Photography, The Digital Darkroom (Holzmann), 895
 BiConsumer functional interface, 636, 985
 BiFunction functional interface, 636, 973
 Binary
 literals, 42
 numbers and integers, 66–67
 BinaryOperator<T> predefined functional interface, 409, 636, 973
 binarySearch(), 551, 556–557
 bitCount(), 450, 452
 BitSet class, 581–583
 methods, table of, 581–582
 Bitwise operators, 66–74
 Block lambdas, 382, 388–389. *See also Lambda expression(s)*
 BLOCKED, 260
 Blocks of code. *See Code blocks*
 Boolean, 35
 literals, 43
 logical operators, 75–77
 Boolean class, 272, 273, 458–460
 and autoboxing/unboxing, 278
 methods, table of, 460
 boolean data type, 35, 40–41, 43, 48
 and relational operators, 40, 41, 74–75
 booleanValue(), 273, 460
 Border interface, 1040
 BorderFactory class, 1040
 BorderLayout class, 798, 858–859, 1032
 example with insets, 860–861
 BorderPane class, 1107, 1178, 1187
 methods for positioning nodes within a, 1178
 boxed(), 968
 Boxing, 274
 break statement, 84–86, 98–99, 102–106
 and the for-each loop, 98–99
 as form of goto, 104–106
 Buffer class, 690–691
 methods, table of, 690–691
 Buffer, NIO, 690–691
 BufferedInputStream class, 303, 659–661, 711
 BufferedOutputStream class, 303, 659, 661–662, 711
 BufferedReader class, 304, 305, 306–307, 676–677, 969
 BufferedWriter class, 304, 678
 Buffering, double, 889–892
 bulkRegister(), 936
 Button class
 AWT, 798, 836
 JavaFX, 1115, 1130, 1133
 ButtonBase class, 1115, 1133, 1135
 ButtonGroup class, 1041, 1051
 ButtonModel interface, 1024, 1045
 Buttons, 773, 782
 JavaFX, 1115–1116, 1130–1142
 push. *See Push buttons*
 radio. *See Radio buttons*
 Swing, 1032–1033, 1045–1053, 1070
 toggle. *See Toggle button, JavaFX;*
 Toggle button, Swing
 ButtonUI, 1024
 Byte class, 273, 447, 454
 methods defined by, table of, 448
 byte data type, 35, 36–37, 41
 and automatic type conversion, 48
 and automatic type promotion, 50, 69–70, 72–73
 ByteArrayInputStream class, 303, 656–657
 ByteArrayOutputStream class, 303, 658–659
 ByteBuffer class, 691, 700, 701, 704, 720
 get() and put() methods, table of, 692
 Bytecode, 9–10, 12, 13, 16, 23–24, 325, 336, 481
 BYTES, 443, 447, 455
 byteValue(), 273, 442, 443, 444, 448, 449, 450, 452
-
- ## C
- C
 history of, 4–5
 and Java, 3, 5, 7, 11
C Programming Language, The (Kernighan and Ritchie), 4
 C++
 history of, 5–6
 and Java, 3, 7, 11
 C# and Java, 8
 Calendar class, 586, 587, 588–591, 592, 596, 1013
 constants, 590
 methods defined by, table of a sampling of, 588–589
 Call-by-reference, 136, 137
 Call-by-value, 136–137, 138
 call(), 939, 962
 Callable interface, 917, 939–942, 962
 CallSite class, 496
 cancel(), 602, 603, 961
 Canvas class
 AWT, 798, 801, 886, 1209
 JavaFX, 1119–1123
 capacity(), 433, 563, 690
 capacityIncrement Vector data member, 563
 Card layouts, 862–865
 CardLayout class, 798, 862–863
 CaretEvent class, 1043
 Case sensitivity and Java, 23, 25, 32, 188
 case statement, 84–87, 88–89
 Casts, 48–50, 338, 341, 342, 344
 and casting one instance of a generic class into another, 370
 and erasure, 341, 373
 using instanceof with, 322–324
 catch clause(s), 213, 214, 216–219, 222, 224, 232
 displaying exception description within, 218
 and the more precise (final) rethrow feature, 231, 232
 multi-catch feature of, 231–232
 using multiple, 218–219
 and nested try statements, 217, 220
 cbrt(), 478
 ceil(), 478
 CGI (Common Gateway Interface), 10, 1211–1212
 changed(), 1139, 1161
 ChangeListener interface, 1138–1139
 Channel interface, 691–692
 Channel(s), NIO, 690, 691–693. *See also NIO and channel-based I/O*

char data type, 35, 39–40, 42, 61, 415
 and automatic type conversion, 48
 and automatic type promotion, 50
 Character class, 272, 273, 455–458
 and autoboxing/unboxing, 278
 methods, table of various, 457, 459
 support for 32-bit Unicode, 458
 Character(s), 35, 39–40
 basic multilingual plane (BMP), 458
 changing case of, 429–430, 456, 457
 classes (regular expressions), 995, 999
 code point, 458
 escape sequences, 43, 44
 extraction from String objects, 419–420
 formatting an individual, 609
 literals, 43
 supplemental, 458
 Character.Subset class, 458
 Character.UnicodeBlock class, 458
 characteristics(), 527, 528
 CharArrayReader class, 304, 674–675
 CharArrayWriter class, 304, 675–676
 charAt(), 153, 419, 434, 493
 CharBuffer class, 691
 chars(), 493
 CharSequence interface, 413, 430, 435, 493, 994
 Charsets, 416, 693
 charValue(), 373, 455
 Check boxes, 751
 AWT, 776, 782, 840–843
 JavaFX, 1142–1145
 Swing, 1049–1051
 and Swing menus, 1081, 1082–1083
 checkAll(), 892
 Checkbox class
 AWT, 798, 840–842
 JavaFX, 1142, 1145
 CheckboxGroup class, 798, 842–843
 CheckboxMenuItem class, 798, 870, 871, 872
 checked... methods, 550, 551–552
 checkedCollection(), 550, 551
 checkedList(), 550, 551
 checkedMap(), 550, 551
 checkedSet(), 550, 551
 checkID(), 892, 895
 CheckMenuItem class, 1172, 1183
 Choice class, 798, 844–846
 Choice controls, 782, 844–848
 ChoiceBox control, 1154
 Class class, 281–282, 283, 285, 286, 340, 458, 460, 473–477, 699, 1001, 1002, 1003
 methods, table of some, 474–475
 .class filename extension, 24, 112
 class keyword, 24, 109
 CLASS retention policy, 281
 Class(es), 109–128
 abstract, 181–184, 185, 199–200
 access levels of, 190–191
 adapter, 791–793
 anonymous, 16. *See also* Inner classes
 character, regular expression, 995, 999
 and code, 23, 109, 190
 in collections, storing user-defined, 529–530
 constructor. *See* Constructor
 controlling access to. *See* Access control
 as a data type, 109, 110, 113, 114, 116, 126
 definition of, 19
 encapsulation achieved through, 126–127
 final, 185
 general form of, 109–110
 generic. *See* Generic class
 inner. *See* Inner classes
 instance of a, 19, 109, 111
 and interfaces, 187, 196, 197–201, 361–362
 libraries, 23, 34
 literal, 283
 member. *See* Member, class
 name and source file name, 23, 24
 nested, 149–151
 packages as containers for, 187, 190, 194
 public, 191
 scope defined by a, 46
 type for bounded types, using a, 347–349
 ClassCastException, 226, 502, 504, 506, 508, 510, 531, 534, 542, 550, 557, 559
 ClassDefinition class, 496
 ClassFileTransformer interface, 496
 ClassLoader class, 477
 classModifiers(), 1005
 ClassNotFoundException, 227, 685
 CLASSPATH, 188, 189, 1008
 –classpath option, 188, 189
 ClassValue class, 493
 clear(), 502, 503, 532, 570, 581, 588, 690, 1166
 Client/server model, 8, 10, 727
 and sockets, 731–734
 clone(), 185, 471–473, 492, 563, 570, 581, 587, 588, 593, 1226
 Cloneable interface, 471–473
 CloneNotSupportedException, 227, 471
 Cloning, potential dangers of, 471–472, 473
 close(), 310, 312–314, 315, 316, 317, 318, 495, 606, 619, 621, 626, 630, 648, 649, 650, 651, 652, 656, 658, 667, 668, 671, 674, 675, 683, 684, 685, 686, 701, 732, 734, 743, 966
 within a finally block, calling, 312–314, 649
 Closeable interface, 310, 316, 626, 648, 651, 654, 665, 667, 668, 670, 679, 691
 Closures, 382
 COBOL, 4
 CODE, 760, 761
 Code base, 764
 Code blocks, 28, 30–31, 45, 82–83
 and the break statement, 104–106
 and scopes, 45, 46–47
 static, 145, 326
 synchronized, 249–250
 Code point, definition of, 458
 Code, unreachable, 108, 219
 CODEBASE, 760
 codePointAt(), 431, 438, 458, 459
 codePointBefore(), 431, 438, 459
 codePointCount(), 431, 438
 codePoints(), 493

collect(), 967, 982–985
Collection interface, 501–504, 969, 971, 975
 methods defined by, table of, 502–503
Collection-view, 499, 531, 571–572
Collection(s), 337, 497–577
 algorithms, 499, 550–556, 561
 into arrays, converting, 503, 513–514
 and autoboxing, 500, 514
 classes, 510–521
 concurrent, 916, 943
 cycling through, 499, 500, 521–529
 dynamically typesafe view of a, 550
 and the for-each version of the for loop, 97, 101, 500, 525–526
Framework. *See* Collections Framework
 generic nature of, 500
 interfaces, 499, 501–510
 and iterators, 499, 500, 504, 521–529
 and legacy classes and interfaces, 561–577
 modifiable versus unmodifiable, 501
 and primitive types, 442, 500, 514
 random access to, 530
 storing user-defined classes in, 529–530
 and the stream API, 577, 965, 969
 stream API stream to obtain a, using a, 982–985
 and synchronization, 510, 550
 and type safety, 500, 550
 when to use, 577
Collections class, 403, 499, 550, 555, 561
 algorithms defined by, table of, 551–555
Collections Framework, 97, 101, 274, 497–577
 advantages of generics as applied to the, 337, 500
 JDK 5 changes to, 500
 legacy classes and interfaces, 561–577
 and method inferences, 402
 overview, 498–499
Collector interface, 982
Collectors class, 982
Color class, AWT, 798, 815–818, 1207, 1209
 constants, 754–755
Color class, JavaFX, 1121, 1166
Combo box, JavaFX, 1151–1154
 enabling users to edit a, 1151, 1153–1154
Combo boxes, Swing, 1061–1063
ComboBox class, 1151
ComboBoxBase class, 1151
ComboBoxModel interface, 1061
Comment, 24–25
 documentation, 32–33, 1235–1241
Common Gateway interface (CGI), 10, 1211–1212
commonPool(), 951, 954
Compact profiles, 336
Comparable interface, 358, 361, 423, 493–494, 586, 645
Comparable<Path> interface, 694
Comparator interface, 403, 404, 501, 536, 539, 542, 971
comparator(), 506, 520, 534
Comparators, 518, 519, 520, 539, 540, 542–550
 using a lambda expression with, 546–547
compare(), 403–404, 443, 444, 448, 449, 450, 452, 460, 542, 544–545, 971
compareAndSet(), 917, 946
compareTo(), 269, 270, 423–424, 443, 445, 448, 449, 450, 452, 456, 460, 492, 493, 545, 587, 645
compareToIgnoreCase(), 424
compareUnsigned(), 450, 452
comparing(), 544
comparingByKey(), 536
comparingByValue(), 536
Compilation unit, 23
compile(), 994
Compiler class, 481
Compiler, Java, 23–24
 and main(), 25
Component class, 749, 751, 754, 755, 757, 771, 781, 788, 798, 800–801, 805, 811, 822, 834, 856, 883, 886, 1024, 1025, 1028, 1029, 1036, 1037, 1071
ComponentAdapter class, 792
componentAdded(), 783
ComponentEvent class, 772, 774, 775, 781
componentHidden(), 783
ComponentListener interface, 782, 783, 792
componentMoved(), 783
componentRemoved(), 783
componentResized(), 783
Components, AWT, 1021–1022, 1024
 lightweight versus heavyweight, 883
 and overriding paint(), 882–883
Components, Swing, 1024–1025, 1041–1068
 architecture, 1023–1024
 class names for, table of, 1024–1025
 heavyweight, 1025
 lightweight, 1022, 1041
 painting, 1036–1040
 and pluggable look and feel, 1022–1023
 and tabbed panes, 1053–1055
componentShown(), 783
ComponentUI, 1024
compute(), 949–950, 954, 958, 960, 963
concat(), 427
Concurrency utilities, 14, 915–964
 versus traditional multithreading and synchronization, 964
Concurrent API, 915–916
 packages, 916–917
Concurrent collection classes, 916, 943
Concurrent program, definition of, 915
ConcurrentHashMap class, 917, 943
ConcurrentLinkedDeque, 943
ConcurrentLinkedQueue class, 917, 943
ConcurrentSkipListMap class, 943
ConcurrentSkipListSet class, 943
Condition class, 944
connect(), 732
Console class, 680–682
 methods, table of, 681
Console I/O, 26, 93, 301, 305–309, 680–682
console(), 467, 680
const keyword, 34
Constants, 32
Constructor class, 282, 285, 286, 496, 1002
Constructor reference, 404–408
 for an array, 408
 to generic classes, 405–408

- C**
- Constructor(s), 114, 121–124
 - in class hierarchy, order of execution of, 174–175
 - default, 114, 123
 - enumeration, 267–269
 - factory methods versus overloaded, 729
 - generic, 359
 - object parameters for, 135–136
 - overloading, 132–134
 - parameterized, 123–124
 - reference. *See* Constructor reference(s)
 - and super(), 167–170, 174, 336
 - this() and overloaded, 334–336
 - constructorModifiers(), 1005
 - Consumer<T> predefined functional interface, 409, 494, 526, 636, 972, 987
 - Container class, 749, 798, 801, 834, 856, 858, 860, 1024, 1025, 1029, 1037, 1071
 - Container, JavaFX, 1106–1107
 - Container(s), Swing, 1024
 - lightweight versus heavyweight, 1025
 - panes, 1025. *See also* Content pane
 - top-level, 1024, 1025
 - ContainerAdapter class, 792
 - ContainerEvent class, 772, 774–775
 - ContainerListener interface, 782, 783, 792
 - Containment hierarchy, 1024, 1025
 - contains(), 431, 502, 503, 516, 563, 570
 - containsAll(), 502, 503
 - Content pane, 1025, 1028–1029, 1033, 1040, 1054, 1056, 1064, 1067
 - default layout manager of JFrame, 1029, 1032
 - ContentDisplay enumeration, 1129–1130, 1133
 - contentEquals(), 431
 - Context switching, 233, 248, 261
 - rules for, 235
 - ContextMenu class, 1172–1173, 1185–1188
 - ContextMenuEvent class, 1188
 - continue statement, 106–107
 - Control class, 1107, 1112, 1115, 1170
 - Control statements. *See* Statements, control
 - Control(s), AWT, 833, 834–855
 - action events, using an anonymous inner class or lambda expression to handle, 839–840
 - definition of an, 833
 - fundamentals, 834–835
 - Control(s), JavaFX, 1107, 1112, 1114, 1125–1170
 - adding an image to a, 1125, 1128–1133
 - adding a tooltip to a, 1170
 - disabling, 1170
 - and effects and transforms, 1164–1170
 - convert(), 942, 943
 - ConvolveOp built-in convolution filter, 910
 - Convolution filters, 902, 907, 910
 - Cookie class, 1222, 1224–1225, 1230, 1232
 - methods, table of, 1226
 - CookieHandler class, 741
 - CookieManager class, 741
 - CookiePolicy interface, 741
 - Cookies, 741, 1224–1225
 - example servlet using, 1230–1232
 - CookieStore interface, 741
 - copy(), 696, 708–709
 - copyOf(), 521, 522, 557
 - copyOfRange(), 557–558
 - CopyOnWriteArrayList class, 917, 943
 - CopyOnWriteArraySet class, 943
 - copySign(), 480
 - cos(), 38, 477
 - cosh(), 477
 - count(), 967, 973, 990
 - countDown(), 924–925
 - CountDownLatch class, 916, 917, 923–925
 - CountedCompleter class, 948
 - countStackFrames(), 482
 - createImage(), 886, 895, 896, 900
 - createLineBorder(), 1040
 - CropImageFilter class, 899, 900–901
 - Currency class, 604–605
 - methods, table of, 604
 - currentThread(), 237, 482
 - currentTimeMillis(), 467, 469
 - CustomMenuItem class, 1172
 - CyclicBarrier class, 916, 917, 925–927
-
- D**
- Data types, 27. *See also* Type(s); Types, primitive
 - DatagramPacket class, 742, 743–745
 - methods, list of some, 744
 - Datagrams, 728, 742–745
 - server/client example, 744–745
 - DatagramSocket class, 692, 742–743, 744–745
 - DataInput interface, 667, 668, 669, 685
 - DataInputStream class, 303, 667, 668–669
 - DataOutput interface, 667, 668, 669, 683
 - DataOutputStream class, 303, 667–669
 - Date and time. *See* Time and date; Time and date API
 - Date class, 586–588, 1010
 - methods, table of, 587
 - DateFormat class, 587, 596, 1009–1011, 1015
 - DateTimeFormatter class, 1015–1017
 - Deadlock, 255–257, 482, 1030
 - decode(), 448, 449, 450, 452, 820
 - Decoder class, 635
 - Decrement operator (–), 30, 61, 64–65
 - decrementAndGet(), 917, 946
 - decrementExact(), 480
 - deepEquals(), 558
 - deepHashCode(), 560
 - deepToString(), 560
 - default
 - clause for annotation member, 287–288
 - to declare a default interface method, using, 208 statement, 84–85
 - DefaultMutableTreeNode class, 1064
 - defaults Properties instance variable, 572–573
 - DelayQueue class, 943
 - Delegation event model, 770–771, 1115
 - and Beans, 1202
 - and event listeners, 770, 771, 782–785
 - and event sources, 770–771, 781–782
 - and Swing, 1030
 - using, 785–791
 - delete operator, 125

- d**
 delete(), 436–437, 644, 696
 deleteCharAt(), 436–437
 deleteOnExit(), 645
 delimiter(), 629
 Delimiters, 579–580
 - Scanner class, 621, 628–629
@Deprecated built-in annotation, 290, 292
D
 Deque interface, 501, 509–510, 515, 520
 - methods, table of, 509–510
 descendingIterator(), 507, 509, 510
 destroy(), 461, 464, 482, 484, 749, 751, 753, 756, 1033, 1212, 1215, 1217
 destroyForcibly(), 461
 Destuctors versus finalize(), 126
 Dialog boxes, 876–882
 - file, 880–882
 Dialog class, 798, 876
 Diamond operator (<>), 372–373
 Dictionary class, 498–499, 561, 568–569
 - abstract methods, table of, 569
 digit(), 456
 Dimension class, 798, 802, 814
 - reflection example using the, 1002–1003
 Directories as File objects, 643, 645–646
 - creating, 648
 Directory, listing the contents of a
 - using list(), 645–647
 - using listFiles(), 647–648
 - using NIO, 714–717
 Directory tree, obtaining a list of files in a, 717–718
 DirectoryStream<Path> class, 714
 DirectoryStream.Filter interface, 716
 dispose(), 876
 distinct(), 990
 divideUnsigned(), 450, 452
 DLL (dynamic link library), 326, 328
 do-while loop, 90–93
 Document base, 764
 Document interface, 1043
 Document/view methodology, 601
@Documented built-in annotation, 290, 291
doDelete(), 1226, 1227
doGet(), 1226, 1227, 1228
doHead(), 1226, 1227
Domain name, 728, 729
Domain Naming Service (DNS), 728
doOptions(), 1226
doPost(), 1227, 1229
doPut(), 1227
DosFileAttributes class, 699, 714
DosFileAttributeView interface, 699
Dot operator (.), 111, 117–118, 146, 170, 188, 194, 211
doTrace(), 1227
Double buffering, 889–892
Double class, 272, 273, 442–446, 454
 - methods, table of, 444–446
 - double data type, 35, 38–39, 42
 - and automatic type conversion, 48
 - and automatic type promotion, 50–51
 DoubleAccumulator class, 947
 DoubleAdder class, 947
 DoubleBinaryOperator functional interface, 560
 DoubleBuffer class, 691
 doubles(), 597–598
 DoubleStream interface, 968, 969
 DoubleSummaryStatistics class, 635
 doubleToLongBits(), 445
 doubleToRawLongBits(), 445
 doubleValue(), 273, 347, 442, 443, 445, 448, 449, 450, 452
 - drawArc(), 812, 813–814
 - drawImage(), 887, 890, 891–892
 - drawLine(), 811, 813–814, 1036
 - drawOval(), 812, 813–814
 - drawPolygon(), 813–814
 - drawRect(), 812, 813–814, 1036
 - drawRoundRect(), 812, 813–814
 - drawString(), 319, 748, 754, 756, 767, 825, 832
 Duration class, 1018
 Dynamic link library (DLL), 325–326, 328
 Dynamic method
 - dispatch, 178–181
 - lookup, 198
 - resolution, 196, 198, 199, 204

E

-
- E**
 E (Math constant), 477
 Early binding, 184
 echoCharIsSet(), 852
 Eclipse IDE, 1212, 1213
 Edit control, 852
 Effect class, 1165
 Effects, 1165–1166
 - list of some built-in, 1165
 - program demonstrating, 1167–1170
 element(), 508
 elementAt(), 563
 elementCount Vector data member, 563
 elementData Vector data member, 563
 elements(), 563, 569, 570
 ElementType enumeration, 291, 496
 ElementType.TYPE_USE, 293, 206
 else, 81–84
 empty(), 567, 584, 585
 EMPTY_LIST static variable, 555
 EMPTY_MAP static variable, 555
 EMPTY_SET static variable, 555
 EmptyStackException, 567, 568
 Encapsulation, 5, 18–19, 20, 22–23, 126–127, 167
 - and access control, 141
 - and scope rules, 46
 Encoder class, 635
 end(), 994
 endsWith(), 422, 694
 ensureCapacity(), 433, 513, 563
 entrySet(), 531, 532, 536, 539, 572
 enum, 263, 492, 521, 541
 Enum class, 269, 492
 - methods, table of, 492
 EnumConstantNotPresentException, 226
 enumerate(), 482, 485, 488
 Enumeration interface, 561–562, 564–566, 568, 579, 580, 663

- Enumeration(s), 14, 263–272, 492, 566
 `= =` relational operator and, 264, 269
 as a class type in Java, 263, 267–269
 constants, 263, 264, 267, 268, 269
 constructor, 267–269
 restrictions, 269
 values in switch statements, using, 264–265
 variable, declaring an, 264
- EnumMap class, 537, 541–542
- EnumSet class, 511, 521
 factory methods, table of, 522
- Environment properties, list of, 470
- equals(), 153, 185, 186, 269, 270, 280, 420, 443, 445, 448, 449, 450, 452, 458, 460, 471, 491, 492, 502, 504, 532, 537, 542, 545, 558, 569, 581, 584, 587, 588, 730, 820
 versus `= =`, 492–493
- equalsIgnoreCase(), 420
- Erasure, 341, 373–376
 and ambiguity errors, 375–377
 bridge methods and, 374–375
- err, 304, 305, 467
- Error class, 214–215, 223, 230, 680
- Errors
 ambiguity, 375–376
 assertions to check for, using, 328–330
 autoboxing/unboxing and prevention of, 278–279
 automatic type promotions and compile-time, 50
 compile-time versus run-time, 344
 generics and prevention of, 342–344
 raw types and run-time, 364
 run-time, 12, 213, 322. *See also* Exception
 handling
 unreachable code, 108, 219
- Event
 and applets, 751, 769
 bubbling, 1115
 change, 1138–1139, 1147, 1151, 1164
 definition of an, 770
 design patterns for a Java Bean, 1202
 dispatch chain, 1115
 dispatching thread and Swing, 1029–1030, 1033, 1034, 1035
 driven programs, 769, 1029
 filter, 1115
 listeners, 770–771, 782–785
 loop with polling, 234, 251
 model, delegation. *See* Delegation event model
 multicasting and unicasting, 771, 1202
 sources, 770–771, 781–782
 timestamp, 773
- Event class, 1115
- Event handling, 751, 769–795
 and adapter classes, 791–793
 event classes, 771–781
 and inner classes, 151, 793–795, 839, 840
 and JavaFX, 1112, 1114–1119
 keyboard, 788–791
 and lambda expressions, 839–840, 1033
 mouse, 785–788
 and Swing, 771, 1022, 1030–1033
See also Delegation event model
- Event listener interfaces, 782–785
 and adapter classes, 791–793
 table of commonly used, 782
- EventHandler interface, 1115, 1118, 1119
- EventListener interface, 635
- EventListenerProxy class, 635
- EventObject class, 635, 771–772, 1115
- EventSetDescriptor class, 1202, 1204, 1206
- Exception, definition of an, 213
- Exception class, 214–215, 227, 229, 230
- Exception classes
 and generics, 379
 hierarchy of the built-in, 214–215
- Exception handling, 12, 93, 102, 213–232, 312, 313–314, 315
 block, general form of, 214
 and chained exceptions, 13, 230–231
 and creating custom exceptions, 227–229
 and the default exception handler, 215–216, 222
 and lambdas, 394–395
 and the more precise (final) rethrow feature, 231, 232
 multi-catch, 231–232
 and suppressed exceptions, 228, 318
 and uncaught exceptions, 215–216, 495, 762
- Exceptions, built-in, 226–227
 checked, table of, 227
 run-time, constructors for, 223
 unchecked, table of, 226
- Exceptions, I/O, 649
- exchange(), 927, 928–929
- Exchanger class, 916, 917, 927–929
- exec(), 460, 461–462, 464, 465
- execute(), 937, 951, 960–961
- Execution point, 491
- Executor interface, 917, 937
- Executors, 916
 using, 937–939
- Executors class, 917, 937
- ExecutorService interface, 917, 937, 940
- exists(), 643, 696, 712
- exit(), 1078, 1180
- exitValue(), 461, 464
- exp(), 478
- expm1(), 478
- Expression lambda, 387. *See also* Lambda expression(s)
- Expressions
 and autoboxing/unboxing, 276–277
 automatic type promotion in, 50–51
 regular. *See* Regular expressions
- extends, 161, 163, 206, 347, 352, 365
 and bounded wildcard arguments, 353, 356
- Externalizable interface, 683, 1203

F

-
- false, 34, 40, 41, 43, 75, 76, 123
 FALSE, 458
 FAT file system, 699, 714
 Field class, 282, 285, 286, 496, 1002
 Field, final, 146–147
 fieldModifiers(), 1005

fields array, 588
 File attribute(s)
 File to access, using, 642–645
 interfaces, 698–699
 NIO to access, using, 699, 712–714
 view interfaces, 699
 File class, 620, 642–648, 665, 679, 712
 instance into a Path instance, converting a,
 645, 695
 methods, 643–645, 652
 file(), 465
 File(s)
 to a buffer, map a, 693, 704–705, 707–708,
 721–723, 724–725
 close() to close a, using, 310, 312–314, 315,
 318, 656
 I/O, 309–318, 642–648. *See also* NIO; NIO and
 channel-based I/O
 path to a, obtaining a, 698, 700
 pointer, 699, 670
 source, 23–24
 system, accessing the, 700
 try-with-resources to automatically close a, using,
 310, 315–318, 656
 FileChannel class, 692, 693, 701, 704, 705–706, 720
 FileDialog class, 798, 880–882
 FileFilter interface, 648
 FileInputStream class, 303, 309–310, 652–654, 692, 720,
 721, 722, 723
 FilenameFilter interface, 646–647
 FileNotFoundException, 310, 313, 649, 652, 654, 672
 FileOutputStream class, 303, 309–310, 314, 654–656,
 692, 723
 FileReader class, 304, 620, 672
 Files class, 642, 693, 695–697, 699, 708, 709, 712,
 714, 717
 methods, table of a sampling of, 696–697
 FileStore class, 700
 FileSystem class, 700
 FileSystems class, 700
 FileVisitor interface, 717–718
 FileVisitResult enumeration, 718
 FileWriter class, 304, 673–674
 fill(), 552, 558
 fillArc(), 812, 813–814
 fillInStackTrace(), 228
 fillOval(), 812, 813–814, 1120
 fillPolygon(), 813
 fillRect(), 812, 813–814, 1120
 fillRoundRect(), 812, 813–814
 fillText(), 1120
 filter(), 584, 586, 967, 972–973, 980
 FilteredImageSource class, 895, 899–900
 FilterInputStream class, 303, 659, 668
 FilterOutputStream class, 303, 659, 667
 FilterReader class, 304
 FilterWriter class, 304
 final, 146–147
 to prevent class inheritance, 185
 to prevent method overriding, 184
 Finalization, 126
 finalize(), 126, 185, 471
 finally block, 213, 214, 224–226, 312–313, 649
 find(), 695, 994, 996–997, 998
 findInLine(), 629–630
 findWithinHorizon(), 630
 Finger protocol, 735
 fire(), 1136, 1138, 1139, 1175
 first(), 506, 863
 firstElement(), 563
 firstKey(), 532
 flatMap(), 584, 586, 982
 flatMapToDouble(), 982
 flatMapToInt(), 982
 flatMapToLong(), 982
 flip(), 581, 690, 707
 Float class, 272, 273, 442–444, 446, 454
 methods, table of, 443–444
 float data type, 35, 38, 42
 and type promotion, 50–51
 Floating-point(s), 35, 38–39
 literals, 42–43
 FloatBuffer class, 691
 floatValue(), 273, 442, 443, 445, 448, 449, 450, 452
 floor(), 478, 507
 floorDiv(), 478
 floorMod(), 478
 FlowLayout class, 799, 856–857, 1032
 FlowPane class, 1107, 1110, 1111, 1118–1119, 1187
 flush(), 606, 648, 652, 661, 671, 681, 683, 684
 Flushable interface, 648, 651, 654, 665, 667, 670,
 679, 680
 FocusAdapter class, 792
 FocusEvent class, 772, 774, 775
 focusGained(), 783
 FocusListener interface, 782, 783, 792
 focusLost(), 783
 Font class, AWT, 799, 820, 821, 822, 824
 methods, table of some, 820
 Font class, JavaFX, 1120
 Font(s), 819–825
 creating and selecting, 822–824
 determining available, 821–822
 information, obtaining, 824
 metrics to manage text output, using, 825–832
 terminology used to describe, 825
 FontMetrics class, 799, 825–827
 methods, table of some, 826
 for loop, 29–30, 40, 93–102
 enhanced. *See* For-each version of the for loop
 variations, 96–97
 For-each version of the for loop, 14, 93, 97–101
 and arrays, 97–101
 and the break statement, 98–99
 and collections, 97, 101, 500, 501, 525–526
 general form, 97
 and the Iterable interface, 494, 500, 525
 and maps, 531
 forceTermination(), 936
 forDigit(), 456
 forEach(), 494, 532, 967, 968, 972, 977
 forEachOrdered(), 977
 forEachRemaining(), 522, 523, 526, 527–528, 988

- Fork/Join Framework, 15, 235, 261, 636, 915–916, 917, 937, 947–964
 advantages to using the, 947–948
 classes, main, 948–951
 tips for using the, 963–964
- Fork/Join Framework divide-and-conquer strategy, 950, 951–954, 963
 and the sequential processing threshold
 interaction with the level of parallelism, 955–958
- Fork/Join Framework tasks, 949
 asynchronous execution of, 960–961
 cancelling, 961
 completion status of, 961
 and the parallelism level, 950, 963
 restarting, 961
 starting, 951, 960–961
 and subtasks, 952
 that do not return a result, 948, 949, 958
 that return a result, 948, 950, 958
- `fork()`, 949, 951, 954, 958, 960, 962
- ForkJoinPool class, 917, 937, 948, 949, 950–951, 952, 954, 955, 958, 960–961, 963
 common pool, 950, 951, 954–955, 958, 963
 and work stealing, 951, 962
- ForkJoinTask class, 917, 948–949, 950, 951, 954, 955, 961, 962, 963–964
- Format flags, 614–617
- Format specifiers (conversions), 605, 605–619
 argument index with, using an, 618–619
 and format flags, 614–617
 and specifying minimum field width, 612–613
 and specifying precision, 614
 suffixes for the time and date, table of, 611
 table of, 607–608
 uppercase versions of, 617–618
- `format()`, 431, 606, 607–608, 618, 666, 667, 679, 680, 681, 1009–1010, 1015
- FormatStyle enumeration, 1015, 1016
- Formattable interface, 635
- FormattableFlags class, 635
- Formatted input, using Scanner to read, 620–630
- Formatter class, 605–620, 666
 closing an instance of the, 619–620
 constructors, 605–606
 methods, table of, 606
 See also Format specifiers
- `forName()`, 474, 1002
- FORTRAN, 4, 5
- Frame class, 799, 800, 801, 802, 803, 805
- Frame window(s), 802–810
 creating a stand-alone, 809–810
 handling events in, 805–809
 within an AWT-based applet, creating, 803–804
- Frank, Ed, 6
- `freeMemory()`, 462–563
- `from()`, 465, 587, 592
- FTP (File Transfer Protocol), 728, 735
- Function<T,R> predefined functional interface, 409, 543, 637, 978
- Functional interfaces, 16, 292, 382, 383–384, 386, 393
 and their abstract methods, table of, 636–639
 generic, 389–391
 predefined, 408–409
- @FunctionalInterface built-in annotation, 290, 292
- Future interface, 917, 940–942
- FXCollections class, 1146, 1151
-
- ## G
- Garbage collection, 12, 125, 126, 139, 462–463, 496
 and images, 893
- `gc()`, 462, 463, 467
- Generic class
 and casting, 370
 example program with one type parameter, 338–342
 example program with two type parameters, 345–346
 general form of a, 346
 hierarchies, 364–372
 and instanceof, 368–370
 overriding methods in a, 371–372
 and raw types, 362–364
 and type inference, 372–373
- Generic constructors, 359
- Generic interfaces, 338, 360–362
 and classes, 361–362
- Generic method, 338, 350, 356–359, 377
- Generics, 13, 14, 274, 337–379
 and annotations, 299
 and ambiguity errors, 375–376
 and arrays, 377–379
 and casts, 338, 341, 344
 and the Collections Framework, 337, 500
 and compatibility with pre-generics code, 362–364, 373
 and exception classes, 379
 restrictions when using, 377–379
 type checking and, 341, 342–344, 363, 379
- GenericServlet class, 1215, 1217, 1220, 1225
- `get()`, 504, 505, 516, 531, 532, 537, 568, 569, 570, 581, 584, 585, 589, 638, 698, 700, 704, 705, 723, 741, 940, 942, 946, 971, 972, 985
 and buffers, 691, 692, 703, 721
- `getActionCommand()`, 773, 837, 847, 1045, 1051, 1052, 1078
- `getActiveThreadCount()`, 963
- `getAddListenerMethod()`, 1206
- `getAddress()`, 730, 744
- `getAdjustable()`, 774
- `getAdjustmentType()`, 774, 850
- `getAlignment()`, 835
- `getAllByName()`, 729, 730
- `getAllFonts()`, 821
- `getAndSet()`, 917, 946, 947
- `getAnnotation()`, 282, 286, 297–298, 474, 489
- `getAnnotations()`, 285–286, 474, 489
- `getAnnotationsByType()`, 287, 298–299, 474, 489
- `getApplet()`, 761, 765
- `getAppletContext()`, 749, 765
- `getArrivedParties()`, 936
- `getAsDouble()`, 586
- `getAscent()`, 826
- `getAsInt()`, 586

getAsLong(), 586
 getAttribute(), 1218, 1219, 1225, 1232
 getAttributeNames(), 1225, 1232
 getAudioClip(), 749, 765, 767
 getAvailableFontFamilyNames(), 821
 getBackground(), 755
 getBeanInfo(), 1206
 getBlue(), 816
 getButton(), 779
 getByAddress(), 730
 getByName(), 729
 getBytes(), 420, 654
 getCalendarType(), 592
 getCause(), 228, 230–231
 getChannel(), 692, 720, 721, 722, 723
 getChars(), 419–420, 434–435, 673
 getChild(), 775
 getChildren(), 1112–1113, 1160, 1161
 getClass(), 185, 186, 281–282, 340, 471, 473, 476, 1003
 getClickCount(), 779
 getCodeBase(), 749, 764
 getColor(), 817
 getCommonPoolParallelism(), 958
 getComponent(), 774, 1084, 1085, 1086
 getConstructor(), 282, 474
 getConstructors(), 474, 1002
 getContainer(), 775
 getContenLengthLong(), 737, 738
 getContentType(), 1029, 1032
 getContents(), 633
 getContentType(), 737, 738
 getCookies(), 1223, 1231
 getDate(), 737, 738
 getDateInstance(), 1009–1010
 getDateTimeInstance(), 1011
 getDeclaredAnnotation(), 286
 getDeclaredAnnotations(), 286, 474, 489
 getDeclaredAnnotationsByType(), 287, 298, 474, 489
 getDeclaredMethods(), 475, 1003
 getDefault(), 593, 595
 getDescent(), 826
 getDirectionality(), 458
 getDirectory(), 881
 getDisplayCountry(), 595
 getDisplayLanguage(), 595
 getDisplayName(), 595, 604
 getDocumentBase(), 749, 764
 getEchoChar(), 852
 getErrorStream(), 461
 getEventSetDescriptors(), 1202, 1209
 getExpiration(), 737, 738
 getExponent(), 480
 GetField inner class, 685
 getField(), 282, 475
 GetFieldID(), 327
 getFields(), 475, 1002
 getFile(), 881
 getFileAttributeView(), 699
 getFiles(), 882
 getFirst(), 509, 515
 getFont(), 820, 824, 826, 1120
 getForeground(), 755
 getForkJoinTaskTag(), 962
 getFreeSpace(), 645
 getGraphics(), 757, 811, 890
 getGraphicsContext2D(), 1120
 getGreen(), 816
 getHeaderField(), 737
 getHeaderFields(), 737, 741
 getHeight(), 826, 1037
 getHostAddress(), 730
 getHostName(), 731
 getHour(), 1017
 getHvalue(), 1158
 getIcon(), 1042
 getID(), 483, 593, 772
 getImage(), 749, 765, 886–887
 getInetAddress(), 732, 743
 getInitParameter(), 1218
 getInitParameterNames(), 1218
 getInputStream(), 461, 464, 732, 737
 getInsets(), 860, 1037
 getInstance(), 589, 591, 604
 getInteger(), 450
 GetIntField(), 327
 getItem(), 777, 844, 847, 872, 1048, 1050
 getItemCount(), 844, 847
 getItems(), 1174, 1179, 1189
 getItemSelectable(), 777, 847
 getKey(), 537, 539
 getKeyChar(), 778
 getKeyCode(), 778
 getLabel(), 836, 840, 871
 getLast(), 509, 515
 getLastModified(), 737, 738, 1227
 getLeading(), 826
 getListenerType(), 1206
 getLocale(), 632, 749
 getLocalGraphicsEnvironment(), 821
 getLocalHost(), 729
 getLocalizedMessage(), 228
 getLocalPort(), 732, 743
 getLocationOnScreen(), 779
 getLong(), 452
 getMaximum(), 849
 getMenuComponentCount(), 1073
 getMenuComponents(), 1073
 getMenuCount(), 1071
 getMenus(), 1173
 getMessage(), 223, 228
 getMethod(), 282, 284–285, 475, 1206, 1223
 getMethodDescriptors(), 1202
 getMethods(), 475, 1002
 getMinimum(), 849
 getMinimumSize(), 856
 getModifiers(), 773, 776, 1003
 getModifiersEx(), 776
 getMonth(), 1017
 getN() getter method design pattern, 1200, 1201
 getName(), 236, 238, 340, 475, 483, 485, 490, 643, 694, 712, 820, 1004, 1206, 1226, 1231
 getNameCount(), 694
 getNewState(), 781
 GetObjectClass(), 327
 getOffset(), 593, 744
 getOldState(), 781

getOppositeComponent(), 775
 getOppositeWindow(), 781
 getOutputStream(), 461, 464, 732, 1219
 getParallelism(), 958
 getParameter(), 749, 761–762, 1219, 1221, 1228, 1229
 getParameterNames(), 1219, 1221
 getParent(), 485, 643, 694, 712, 936, 1161
 getPath(), 1063–1064, 1226
 getPhase(), 931
 getPoint(), 778
 getPoolSize(), 963
 getPort(), 732, 743, 744
 getPreciseWheelRotation(), 780
 getPreferredSize(), 856
 getPriority(), 236, 246, 483
 getProperties(), 468, 572
 getProperty(), 468, 470, 573, 574, 575
 getPropertyDescriptors(), 1202, 1203, 1208, 1209
 getQueuedTaskCount(), 962
 getRed(), 816
 getRegisteredParties(), 936
 getRemoveListenerMethod(), 1206
 getRGB(), 817
 getRuntime(), 461, 462
 getScreenX(), 1188
 getScreenY(), 1188
 getScript(), 595
 getScrollAmount(), 780
 getScrollType(), 780
 getSecurityManager(), 490
 getSelectedCheckbox(), 842
 getSelectedIndex(), 844, 846, 1059
 getSelectedIndexes(), 847
 getSelectedItem(), 844, 846, 1062
 getSelectedItems(), 847, 1150
 getSelectedText(), 852, 854
 getSelectedToggle(), 1142
 getSelectedValue(), 1059
 getSelectionModel(), 1147, 1160
 getServletConfig(), 1217
 getServletContext(), 1218
 getServletInfo(), 1217
 getServletName(), 1218
 getSession(), 1223, 1232
 getSize(), 802, 814, 820
 getSource(), 772, 838, 1052, 1115
 getStackTrace(), 228, 483, 491
 getState(), 259–261, 483, 840, 871
 getStateChange(), 777, 847
 getSubElements(), 1072
 getSuperclass(), 475, 476
 getSuppressed(), 228, 318
 getSurplusQueuedTaskCount(), 962
 getTarget(), 1180
 getText(), 835, 852, 854, 1042, 1044, 1045, 1050,
 1154, 1180
 getTimeInstance(), 1010–1011
 getTransforms(), 1166
 getUnarrivedParties(), 936
 getTotalSpace(), 645
 getUsableSpace(), 645
 getValue(), 537, 539, 774, 849, 1226, 1231,
 1151–1152, 1161

getVvalue(), 1158
 getWheelRotation(), 780
 getWhen(), 773
 getWidth(), 1037
 getWindow(), 781
 getWriter(), 1215, 1219
 getX(), 778, 1084, 1086
 getxonScreen(), 779, 1084, 1086
 getY(), 778
 getYear(), 1017
 getyOnScreen(), 779
 GIF image format, 885–886, 887
 Glass pane, 1025
 Glassfish, 1212, 1213
 Glob, 715–716
 Glow class, 1165
 program demonstrating, 1167–1170
 Gosling, James, 6
 goto keyword, 34
 Goto statement, using labeled break as form of,
 104–106
 grabPixels(), 897
 Graphical User Interface. *See GUI (Graphical User Interface)*
 Graphics
 and JavaFX retained mode, 1106, 1119
 context, 319, 753, 811
 sizing, 814–815
 Graphics class, 319, 753, 754, 799, 811, 817, 824,
 887, 890
 drawing methods, 811–814
 Graphics2D class, 811
 GraphicsContext class, 1119–1123
 GraphicsEnvironment class, 799, 821
 GregorianCalendar class, 588, 591–592, 596, 1013
 Grid bag layouts, 865–870
 GridBagConstraints class, 799, 866–868
 constraint fields, table of, 866–867
 GridBagLayout class, 799, 865, 866, 868, 870
 gridheight constraint field, 866, 868
 GridLayout class, 799, 861–862
 GridPane class, 1107
 gridwidth constraint field, 866, 868
 Group class, 1107
 group(), 699, 994
 GIU (Graphical User Interface), 301, 319, 321, 797, 833
 applets based on the, 751
 approaches to the, 1105
 effects and transforms to customize the look of a
 JavaFX, using, 1164–1170
 programs, handling events generated by, 769–795
 GZIP file format, 639

H

handle(), 1115, 1118, 1179
 hasCharacteristics(), 527, 528
 Hash code, 516
 Hash table, 516
 hashCode(), 185, 280, 443, 445, 448, 449, 450, 452,
 458, 460, 471, 490, 491, 492, 502, 532, 537, 560, 569,
 581, 584, 587, 820

- Hashing, 516, 517
 HashMap class, 537–539, 540, 541, 569
 HashSet class, 511, 516–517, 969
 - from a stream API stream, obtaining a, 985
 Hashtable class, 511, 561, 569–572, 573
 - and iterators, 571–572
 - legacy methods, table of, 570
 hasMoreElements(), 562, 580
 hasMoreTokens(), 580
 hasNext(), 522, 523, 986, 987
 hasNextX() Scanner methods, 621, 624
 - table of, 622
 Headers, 737
 HeadlessException, 802, 835
 headMap(), 534, 535
 headSet(), 506, 507
 HEIGHT, 760, 761
 Hexadecimals, 41, 42–43
 - as character values, 43
 Hierarchical abstraction and classification, 18
 - and inheritance, 19, 161
 High surrogate char, 458
 highestOneBit(), 450, 452
 Histogram, 897–899
 Hoare, C.A.R., 236
 Holzmann, Gerard J., 895
 HotSpot technology, 10
 HSB (hue-saturation-brightness) color model, 816
 HSBTtoRGB(), 816
 HSPACE, 760, 761
 HTML (Hypertext Markup Language), 1211, 1215
 - file for an applet, 320–321, 748, 760–761
 - and javadoc, 1235, 1236, 1239
 HTMLEditor, 1156
 HTTP, 728, 735
 - GET requests, handling, 1227–1228
 - and HttpURLConnection class, 739
 - port, 728
 - POST requests, handling, 1227, 1229–1230
 - requests, 1211, 1212, 1222, 1227
 - response, 1211, 1212, 1215, 1222, 1224
 - and URLConnection class, 737
 HTTP session
 - stateful, 741
 - tracking, 1232–1233
 HttpCookie class, 741
 HttpServlet class, 1222, 1225, 1227
 - methods, table of, 1226
 HttpServletRequest interface, 1222, 1228, 1229, 1232
 - methods, table of several, 1223
 HttpServletResponse interface, 1222–1223, 1224
 - methods, table of, 1224
 HttpSession interface, 1222, 1223, 1232
 - methods, table of several, 1225
 HttpURLConnection class, 739–741
 - methods, sampling of, 739
 hypot(), 480
-
- I
- Icon interface, 1042
 Icons
 - Swing button, 1045
 - Swing label, 1042
- Identifiers, 24, 32, 34, 44, 45
 IdentityHashMap class, 537, 541
 IEEEremainder(), 480
 if statement, 28–29, 30, 40, 41, 81–84
 - boolean variable used to control the, 82, 278
 - nested, 83
 - and recursive methods, 140
 - switch statement versus, 88–89
 if-else-if ladder, 83–84
 IllegalAccessException, 224, 227
 IllegalArgumentException, 226, 502, 504, 506, 508, 510, 521, 531, 534, 558
 IllegalFormatException, 608
 IllegalMonitorStateException, 226
 IllegalStateException, 226, 502, 510, 994, 1223
 IllegalThreadStateException, 226
 Image class
 - AWT, 799, 885, 886–887, 890, 895, 897
 - JavaFX, 1225–1227
 ImageConsumer interface, 897–899
 ImageFilter class, 899
 ImageIcon class, 1041, 1042
 ImageObserver interface, 887, 888–889, 892
 ImageProducer interface, 886, 895, 897, 899
 imageUpdate(), 888–889
 - bit flags, table of, 889
 Images, 885–913
 - creating, loading, displaying, 886–888
 - double buffering and, 889–892
 - file formats for web, 885–886
 - filters for, 899–912
 - stream model for, 899–900
 Imaging, 885
 ImageView class, 1125–1127, 1128, 1130
 IMG tag, 761
 implements clause, 197
 - and generic interfaces, 361, 362
 import statement, 194–195
 - and static import, 331–334
 in, 304, 305, 464, 467, 620, 680
 Increment operator (++), 30, 61, 64–66
 incrementExact(), 480
 indexOf(), 424–426, 438–439, 504, 505, 563–564
 IndexOutOfBoundsException, 226, 504
 Inet4Address class, 731
 Inet6Address class, 731
 InetAddress class, 729–731, 742
 InetSocketAddress class, 743
 infinity (IEEE floating-point specification value), 446
 inForkJoinPool(), 962
 INHERIT, 465
 InheritableThreadLocal class, 488
 Inheritance, 5, 18, 19–21, 22–23, 142, 144, 161–186
 - and annotations, 299
 - and enumerations, 269
 - final and, 184–185
 - and interfaces, 187, 196, 206–207, 210–211, 212
 - multilevel, 171–174
 - and multiple superclasses, 163, 187
 @Inherited built-in annotation, 290, 291
 init(), 750, 751, 753, 755–756, 759, 788, 792, 793, 803, 832, 1212, 1215, 1217
 - and JavaFX, 1107, 1108, 1110
 - and Swing, 1033, 1035

initCause(), 228, 230
 Inline method calls, 184
 Inner classes, 149–151, 793–794
 anonymous, 795, 839–840, 1052, 1071,
 1085–1086, 1115, 1119
 InnerShadow class, 1165
 program demonstrating, 1167–1170
 InputEvent class, 772, 775–776, 777, 778, 1079
 InputMismatchException, 624
 InputStream class, 302, 303, 305, 620, 650, 651, 652,
 656, 659, 660, 662, 663, 668, 685, 688, 710, 1220
 methods, table of, 651
 objects, concatenating, 663
 InputStreamReader class, 304, 305
 insert(), 435–436, 854, 1072
 insertSeparator(), 1042
 Insets class, 799, 860–861, 1037
 Instance of a class, 19, 109, 111, 114
 See also Object(s)
 Instance variables
 accessing, 111, 116, 117–118, 120
 default values of, 123
 definition of, 19, 110
 hiding, 125
 and interfaces, 207
 static, 145–146
 as unique to their object, 111, 112–113
 using super to access hidden, 170–171
 instanceof operator, 322–324, 530
 and generic classes, 368–370
 Instant class, 587, 1018
 InstantiationException, 227
 Instrumentation interface, 496
 int, 27, 35, 36, 37
 and automatic type conversion, 48
 and automatic type promotion, 50–51, 69–70, 72
 and integer literals, 41
 IntBuffer class, 691
 Integer class, 272, 273, 274, 447, 454–455, 971
 constructors, 273
 methods, table of, 450–451
 Integer(s), 35, 36–38, 66–67
 literals, 41–42
 interface keyword, 187, 196
 and annotations, 280
 Interface methods
 default, 16, 197, 207–211, 381, 383
 static, 211–212
 traditional, 196, 197–198, 383
 Interface(s), 187, 196–212
 functional. *See* Functional interfaces
 general form of, 196–197
 generic. *See* Generic interfaces
 implementing, 197–200
 and the inheritance hierarchy, 196
 inheritance of, 206–207, 211
 member, 200
 methods. *See* Interface methods
 nested, 200–201
 reference variables, 198–199, 204
 types for bounded types, using, 349
 variables, 197, 204–206
 interfaceModifiers(), 1005
 Internet, 3, 6, 7, 8, 12, 16, 727
 addresses, obtaining, 729–731
 addressing scheme, 728
 and portability, 7, 8, 9
 and security, 8–9
 Internet Engineering Task Force (IETF) BCP 47, 595
 Internet Protocol (IP)
 addresses, 728
 definition of, 727
 InterNIC, 732, 734
 InterruptedException, 227, 237–238, 897
 Introspection, 1200–1203, 1206, 1209
 Introspector class, 1205, 1206
 ints(), 597–598
 IntStream interface, 968, 969, 981
 IntSummaryStatistics class, 635
 intValue(), 273, 274, 442, 444, 445, 448, 449, 450, 452
 InvalidPathException, 698
 invoke(), 949, 951, 955, 960
 invokeAll(), 949, 954, 958, 962
 invokeAndWait(), 1030, 1035
 invokeLater(), 1030, 1035
 I/O, 26, 301–318, 641–688
 and applets, 319, 321
 channel-based, 13, 302, 689. *See also* NIO; NIO
 and channel-based I/O
 classes, list of, 641–642
 console, 26, 93, 301, 305–309, 680–682
 error handling, 312–315
 exceptions, 649
 file, 309–318, 642–648
 formatted. *See* I/O, formatted
 interfaces, list of, 642
 new. *See* NIO
 redirection, 465
 streams. *See* Streams, I/O
 I/O, formatted, 14
 format specifiers. *See* Format specifiers
 using Formatter, 605–620. *See also* Formatter class
 using printf(), 155, 666–667, 680
 using Scanner, 620–630. *See also* Scanner class
 io package. *See* java.io package
 IOException, 93, 305, 310, 313, 314, 649, 651, 652,
 656, 662, 670, 673, 683, 684, 685, 695, 714, 717, 732,
 736, 742
 ipadx constraint field, 866, 868
 ipady constraint field, 866, 868
 IPv4 (Internet Protocol, version 4), 728, 729, 730, 731
 IPv6 (Internet Protocol, version 6), 728, 729, 730, 731
 isAbsolute(), 644, 695
 isAlive(), 236, 243–246, 461, 483
 isAltDown(), 776
 isAltGraphDown(), 776
 isAnnotationPresent(), 286, 288, 490
 isBound(), 732, 743, 1206
 isCancelled(), 961
 isClosed(), 732
 isCompletedAbnormally(), 961
 isCompletedNormally(), 961
 isConnected(), 732, 743
 isConstrained(), 1206
 isControlDown(), 776

- isDigit(), 456, 457, 458
 isDirectory(), 645–646, 696, 698
 isEditable(), 852, 854
 isEmpty(), 431, 502, 503, 532, 564, 568, 569, 570, 581
 isEnabled(), 871, 1091
 isExecutable(), 696, 712
 isFile(), 644
 isFinite(), 444, 445
 isHidden(), 645, 696, 699, 712
 isIndeterminate(), 1145
 isInfinite(), 444, 445, 446–447
 isLeapYear(), 592
 isLetter(), 456, 457, 458
 isLightweight(), 883
 isLowercase(), 456, 457
 isMetaDown(), 776
 isMulticastAddress(), 731
 isMultipleMode(), 882
 isNaN(), 444, 445, 446–447
 ISO-Latin-1 character set, 39, 43
 isPopupTrigger(), 779, 1084, 1086
 isPresent(), 584, 585, 971
 isPropertyName(), 1201
 isPublic(), 1003–1004
 isQuiescent(), 963
 isReadable(), 696, 712
 isSelected(), 1048, 1050, 1052, 1072, 1134, 1135, 1145
 isSet array, 588
 isSet(), 589
 isShiftDown(), 778
 isShutdown(), 963
 isTemporary(), 775
 isTerminated(), 963
 isTimeSet, 588
 isUppercase(), 456, 457
 isWhitespace(), 456, 457
 isWritable(), 696, 699, 712
 ItemEvent class, 772, 776–777, 839, 840, 844, 847, 872,
 1048, 1050
 ItemListener interface, 782, 783, 840, 844, 872,
 1048, 1050
 ItemSelectable interface, 777
 itemStateChanged(), 783, 840, 844, 1048, 1050
 Iterable interface, 431, 494, 500, 501, 525, 531, 562
 Iterable<Path> interface, 694, 714
 Iteration statements, 81, 89–102
 Iterator, 499, 500, 504, 521–529
 and maps, 531
 obtaining an, 523, 524
 and PriorityQueue, 520
 and stream API streams, 986–987
 and synchronized collections, 550
 Iterator interface, 499, 501, 521, 523–525, 526, 562
 methods, table of, 522
 iterator(), 494, 502, 504, 523, 714, 966, 986
-
- J**
- J2SE 5, new features of, 13, 14
 JApplet class, 747, 1025, 1033, 1035
 Java
 API packages, table of core, 991–993
 and C, 3, 5, 7, 11
 and C++, 3, 7, 11
 and C#, 8
 design features (buzzwords), 10–13
 history of, 3, 6–8, 13–16
 and the Internet, 3, 6, 7–9, 12, 16, 727
 as interpreted language, 9, 10, 12
 keywords, 33–34
 as a strongly typed language, 10, 11, 35, 41
 versions of, 13–14
 and the World Wide Web, 6, 7, 11
 Java Archive (JAR) files, 639
 Java Beans, 476, 496, 991, 1001, 1199–1209
 advantages of, 1200
 API, 1204–1206
 customizers, 1203
 demonstration program, 1206–1209
 introspection, 1200–1203, 1206, 1209
 properties. *See* Property, Java Bean
 serialization, 1203
 java filename extension, 23
 Java Community Process (JCP), 16
 Java Control Panel, 748
 Java EE SDK, 1212, 1216
 Java Foundation Classes (JFC), 1022
 java (Java application launcher). *See* Application launcher (java)
 Java Native Interface (JNI), 325
 Java Network Launch Protocol (JNLP), 748, 760, 1111
 java package, 188, 189, 194
 Java SE 7, 14–16
 Java SE 8, 15–16
 Java Virtual Machine (JVM), 9–10, 12, 13, 16, 24, 25,
 461, 496
 java.awt package, 301, 319, 747
 java.awt package, 769, 772, 798, 885, 886, 1032
 classes, tables of some, 798–800
 java.awt.Dimension class, reflection example using the,
 1002–1003
 java.awt.event package, 769, 771, 772, 782, 791,
 1030, 1032
 event classes, table of commonly used, 772
 interfaces, table of commonly used, 772
 java.awt.event.InputEvent class. *See* InputEvent class
 java.awt.event.KeyEvent class. *See* KeyEvent class
 java.awt.image package, 885, 895, 899, 910, 913
 java.beans package, 1202, 1204–1206
 classes, table of, 1204–1205
 interfaces, tables of, 1204
 java.io package, 301, 302–304, 310, 316, 641–642, 648,
 689, 712
 classes, list of, 641–642
 interfaces, list of, 642
 java.io.Externalizable interface, 683, 1203
 java.io.IOException. *See* IOException
 java.io.Serializable interface, 682–683, 687, 962, 1203
 java.lang package, 194, 226, 281, 290, 304, 310, 316,
 358, 361, 413, 441–495, 648
 classes and interfaces, list of, 441
 implicit importation of the, 194
 java.lang.annotation package, 280, 290, 297, 495, 496
 java.lang.annotation.RetentionPolicy enumeration,
 281, 496
 java.lang.image package, 897

- java.lang.instrument package, 495, 496
- java.lang.invoke package, 495, 496
- java.lang.management package, 496
- java.lang.ref package, 496
- java.lang.reflect package, 281, 286, 496, 991, 992, 1001
 - classes, table of, 1002
- java.net package, 727, 741
 - classes and interfaces, list of, 728–729
- java.nio package, 302, 641, 645, 689, 690
- java.nio.channels package, 689, 691, 693
- java.nio.channels.spi package, 689
- java.nio.charset package, 689, 693
- java.nio.charset.spi package, 689
- java.nio.file package, 689, 693, 694
- java.nio.file.attribute package, 689, 693, 698
- java.nio.file.spi package, 689, 693
- java.nio.file.WatchService, 719
- java.rmi package, 991, 992, 1006
- java.text package, 991, 993, 1009
- java.time package, 588, 991, 993, 1013, 1018
- java.time.format package, 1013, 1015
- java.util package, 497–498, 561, 579, 769, 771, 971, 986
 - classes, list of top-level, 497–498
 - interfaces defined by, list of, 498
- java.util.concurrent package, 635, 636, 916–917, 942, 948
- java.util.concurrent.atomic package, 635, 636, 916, 917, 946, 947
- java.util.concurrent.locks package, 635, 636, 916, 917, 943, 944, 946
- java.util.function package, 16, 408–409, 526, 543, 560, 579, 635, 636, 972, 973, 978, 985
 - functional interfaces defined by, table of, 636–639
- java.util.jar package, 635, 639
- java.util.List class. *See* List class
- java.util.logging package, 635, 639
- java.util.prefs package, 635, 639
- java.util.regex package, 636, 639, 991, 993
- java.util.spi package, 636, 639
- java.util.stream package, 16, 636, 639, 966, 982
- java.util.zip package, 636, 639
- javac (Java compiler), 23–24, 188, 293, 364, 1112
- avadoc, 1235–1241
 - tags, 1235–1239
 - utility program, 1235, 1239
- JavaFX, 16, 301, 797, 833, 1105–1123
 - event handling, 1112, 1114–1119
 - images, support for, 1125–1127
 - launcher thread, 1112
 - layout panes, 1107, 1110, 1111, 1118–1119, 1178, 1187, 1196
 - menus. *See* Menus, JavaFX
 - nodes. *See* Node(s), JavaFX
 - packages, 1106
 - repainting, 1106, 1119, 1121
 - scene, 1106–1107, 1110, 1111, 1112
 - scene graph, 1107, 1112–1114, 1118, 1119, 1126, 1157, 1196
 - stage, 1106, 1107, 1110, 1112
 - versus Swing, 1106, 1119
- JavaFX application class, 1107–1108
- compiling and running a, 1111–1112
- launching a, 1108
- skeleton, 1108–1111
- thread, 1112
- javafx.application package, 1106, 1107, 1110, 1080
- javafx.beans.value package, 1139
- javafx.collections package, 1113, 1146
- javafx.event package, 1115, 1116
- javafx.geometry package, 1119
- javafx.scene package, 1106, 1110
- javafx.scene.canvas package, 1119
- javafx.scene.control package, 1112, 1115, 1125, 1136, 1142, 1171
- javafx.scene.effect package, 1196
- javafx.scene.image package, 1125
- javafx.scene.input package, 1181
- javafx.scene.layout package, 1106, 1107, 1110
- javafx.scene.paint package, 1121
- javafx.scene.paint.Color class, 1121, 1166
- javafx.scene.shape package, 1123
- javafx.scene.text package, 1120, 1170
- javafx.scene.transform package, 1166, 1196
- javafx.stage package, 1106, 1110
- javafx.stage.PopupWindow, 1185
- javafxpackager tool, 1108, 1112
- javad.exe, 326, 327
- javap, 375
- javax.imageio package, 913
- javax.servlet package, 1215, 1216–1220
 - interfaces and classes, list of core, 1216–1217
- javax.servlet.http package, 1216, 1222–1227
 - interfaces and classes, list of some, 1222
- javax.swing package, 1024, 1026, 1027, 1041, 1063
 - classes, list of, 1024–1025
- javax.swing.event package, 1030, 1043, 1058, 1063
- javax.swing.table package, 1066, 1067
- javax.swing.tree package, 1063
- JButton class, 1025, 1032, 1041, 1045–1047, 1070, 1091
- JCheckBox class, 1041, 1045, 1047, 1049–1051, 1091
- JCheckBoxMenuItem class, 1070, 1081, 1082–1083
- JComboBox class, 1041, 1061–1063
- JComponent class, 1024, 1025, 1033, 1036, 1037, 1041, 1045, 1071, 1081
- JDialog class, 1025, 1101
- JDK 8 (Java SE 8 Development Kit), 15–16, 23
- JFormattedTextField class, 1101
- JFrame class, 1025, 1026, 1027, 1029, 1040, 1072, 1074
- JIT (Just-In-Time) compiler, 10, 12
- JLabel class, 1025, 1026, 1028, 1030, 1036, 1041–1043, 1074
- JLayeredPane class, 1025
- JList class, 1041, 1058–1060
- JMenu class, 1070, 1071, 1072–1073, 1074
 - mnemonic, 1078
- JMenuBar class, 1070, 1071–1072, 1074
- JMenuItem class, 1070, 1071, 1072, 1073, 1074, 1081, 1082
 - accelerator key, 1079, 1080
 - action to create a, using an, 1091
 - and action events, 1073, 1074, 1077
 - mnemonic, 1078, 1079–1080
- jni.h, 327, 328

jni_md.h, 328
 JNLP (Java Network Launch Protocol), 748, 760, 1111
 join(), 236, 243–246, 430–431, 483, 949, 958, 960
 JOptionPane class, 1101
 Joy, Bill, 6
 JPanel class, 1025, 1037, 1040, 1055, 1056
 JPEG image file format, 886, 887
 JPopupMenu class, 1070, 1083–1086, 1091
 and mouse events, 1084, 1085–1086
 JRadioButton class, 1041, 1045, 1047, 1051–1053, 1091
 JRadioButtonMenuItem class, 1070, 1082–1083, 1091
 JRootPane class, 1025
 JScrollPane class, 1025
 JSeparator class, 1070, 1072
 JScrollPane class, 1041, 1056–1057, 1058, 1064, 1066, 1067
 JSpinner class, 1101
 JTabbedPane class, 1041, 1053–1055
 JTable class, 1041, 1066–1068
 JTextComponent class, 1043
 JTextField class, 1041, 1043–1044
 JToggleButton class, 1041, 1045, 1047–1049, 1051
 JToggleButton.ToggleButtonModel class, 1048
 JToolBar class, 1069, 1087–1089
 adding an action to a, 1091
 JTree class, 1041, 1063–1066
 Jump statements, 81, 102–108
 Just In Time (JIT) compiler, 10, 12
 JVM (Java Virtual Machine), 9–10, 12, 13, 16, 24, 25, 461, 482, 496
 JWindow class, 1025

K

Kernighan, Brian, 4
 Key codes, virtual, 777–778, 790
 KeyAdapter class, 792
 Keyboard events, handling, 788–791
 KeyCombination class, 1181
 keyCombination(), 1181
 KeyEvent class, 772, 774, 775, 777–778, 1078, 1079
 KeyListener interface, 782, 784, 788–791, 792
 keyPressed(), 784, 788, 789
 keyReleased(), 784, 788
 keys(), 568, 569, 570
 keySet(), 531, 532, 572, 632, 741
 KeyStroke class, 1079
 keyTyped(), 784, 788, 789
 Keywords, table of Java, 33

L

Label
 AWT standard control, 835–836
 Swing, 1026, 1028, 1041–1043
 used with break statement, 104–106
 used with continue statement, 107
 Label class
 AWT, 799, 835
 JavaFX, 1112, 1128
 Label, JavaFX, 1112–1114
 adding an image to a, 1128–1130

Labeled class, 1112, 1115
 Lambda expression(s), 15–16, 381–396, 408–409
 as arguments, passing, 391–394
 block, 382, 387–389
 body, 382, 387–388
 and comparators, 546–547
 definition of, 382
 and exceptions, 394–395
 and generics, 389
 to handle action events, 839–840, 1033, 1052, 1071, 1115, 1119
 parameters, 382–383
 and the stream API, 965
 target type, 382, 383, 384, 389–390, 391, 393, 395
 and variable capture, 395–396
 Lambda arrow operator (\rightarrow), 16, 61, 382
 last(), 506, 863
 lastElement(), 563, 564
 lastIndexOf(), 424, 425–426, 438–439, 504, 505, 563, 564
 lastKey(), 534
 Late binding, 184
 launch(), 1108, 1110
 Layered pane, 1025
 Layout managers, AWT, 801, 833, 855–870
 default, 833, 855, 856
 Layout panes, JavaFX, 1107, 1110, 1111, 1118–1119, 1178, 1187, 1196
 LayoutManager interface, 856
 Lazy behavior (stream API stream), 968
 length instance variable of arrays, 147–149
 length(), 153, 416, 433, 493, 581
 Lexer (lexical analyzer), 579
 Libraries, class, 23, 24
 Library, compact profiles of the API, 336
 Lindholm, Tim, 6
 LineNumberInputStream deprecated class, 642
 LineNumberReader class, 304
 lines(), 676, 695, 969
 LinkedBlockingDeque class, 943
 LinkedBlockingQueue class, 943
 LinkedHashMap class, 537, 540–541
 LinkedHashSet class, 511, 517–518
 LinkedList class, 511, 515–516
 example program using the, 529–530
 from a stream API stream, obtaining a, 985
 LinkedTransferQueue, 943
 List
 controls, 846–848
 items, 773, 776, 782
 List class, 799, 846, 847, 1113, 1146, 1150, 1151
 List interface, 501, 504, 511, 515, 516, 524, 556, 562, 563, 1173
 from a stream API stream, obtaining a, 982–984
 methods, table of, 505
 List, Swing, 1058–1060
 List view, 1146–1151
 change events, handling, 1147
 multiple selections in a, enabling, 1150–1151
 scrollbars, 1149–1150
 list(), 573, 695
 and directories, 643, 645–647

list(), ThreadGroup, 485, 487
listFiles(), 647–648
ListIterator interface, 501, 521, 524–525, 526
 methods, table of, 523
listIterator(), 505, 524
ListModel, 1058
ListResourceBundle class, 633
ListSelectionEvent class, 1058, 1059, 1067
ListSelectionListener interface, 1058, 1059
ListSelectionModel interface, 1058–1059, 1067
ListView class, 1146–1151
Literals, 32, 41–44
 class, 283
 regular expression, 995
 string, 43–44, 416
load(), 462, 468, 573, 576–577
loadLibrary(), 326, 462, 468
LocalDate class, 1013, 1014, 1015, 1017, 1018
LocalDateTime class, 1013, 1014–1015, 1017–1018
Locale class, 430, 594–596, 1009, 1010
Locale Data Markup Language (LDML), 595
Locale.Builder class, 595
LocalTime class, 1013, 1014, 1017, 1018
Lock interface, 917, 944
 methods, table of, 944
lock(), 917, 944
lockInterruptibly(), 944
Locks, 943–946
log()
 math method, 478
 servlet method, 1218, 1220
log10(), 478
log1p(), 478
Logical operators
 bitwise, 67–69
 Boolean, 75–77
long, 35, 36, 37–38
 and automatic type conversion, 48
 and automatic type promotion, 50
 literal, 41–42
Long class, 272, 273, 447, 454–455
 methods, table of, 452–453
LongAccumulator class, 947
LongAdder class, 947
longBitsToDouble(), 445
LongBuffer class, 691
longs(), 597–598
LongStream interface, 968, 969
longValue(), 273, 442, 444, 445, 448, 449, 450, 452
Look and feels, 1022–1023
lookup(), 1007
loop(), 767
Loop(s), 81
 Boolean object to control, using a, 278
 continue statement and, 106–107
 do-while, 90–93
 for. *See* for loop
 infinite, 96–97, 103
 nested, 102, 104, 105–106
 with polling, event, 234, 251
 while, 89–90
Low surrogate char, 458
lowestOneBit(), 450, 452

M

main(), 25–26, 110, 142, 145
 and applets, 26, 110, 320, 321, 748
 and the java application launcher, 25
 and command-line arguments, 25, 154–155
 and Swing programs, 1029–1030
 and windowed applications, 809–810
main (default name of main thread), 238
makeGUI(), 1035
MalformedURLException, 735
Map interface, 531–533, 534, 536, 537, 538, 541, 568, 569, 570, 571–572
 methods, table of, 532–533
map(), 584, 586, 693, 704, 705, 707, 722, 724
 and stream API streams, 967, 978–981
Map(s), 499, 530–542
 classes, 537–542
 collection-view of a, obtaining a, 499, 531
 flat, 982
 interfaces, 531–537
 and stream API streams, 978–982
 submaps of, 534
Map.Entry interface, 531, 536, 539
 methods, table of non-static, 537
MapMode.PRIVATE, 704
MapMode.READ_ONLY, 704
MapMode.READ_WRITE, 704, 707
MappedByteBuffer class, 691, 704
mapToDouble(), 981
mapToInt(), 981–982
mapToLong(), 981
mark(), 651, 652, 657, 660, 663, 671, 676, 691
markSupported(), 651, 660, 663, 670, 671, 676
Matcher class, 993, 994–995, 996, 997, 999, 1001
matcher(), 994
matches(), 431, 994, 996, 1001
Math class, 45, 131, 477–481
 rounding methods, table of, 478–479
 and static import example, 331–333
max(), 403–404, 444, 445, 450, 452, 479, 553, 556, 967, 972
MAX_EXPONENT, 443
MAX_PRIORITY, 246, 482
MAX_RADIX, 455
MAX_VALUE, 443, 447, 455
MediaTracker class, 799, 885, 892–895
Member, class, 19, 110
 access and inheritance, 163–164
 access, table of, 191
 controlling access to, 141–144
 static, 145–146
Member interface, 496, 1001
Memory
 allocation using new, 52, 53, 113–114
 deallocation, 125
 leaks, 310, 315, 649
 management, in Java, 11–12, 125
 and the Runtime class, 462–463
MemoryImageSource class, 895–896, 897, 899
Menu bars and AWT menus, 833, 870–876
 action command string of, 872
 and events, 872

- Menu class
 - AWT, 799, 870, 871
 - JavaFX, 1172, 1173, 1174, 1175, 1179
- Menu item as an event source, AWT, 773, 776, 782
- Menu(s), JavaFX, 1171–1196
 - accelerator keys, 1171, 1180–1181
 - check menu items, 1172, 1183–1185
 - classes, table of core, 1172
 - context menu, 1171, 1172–1173, 1185–1188
 - demonstration program, 1191–1196
 - events, handling, 1172, 1175, 1179–1180
 - and images, 1174, 1182–1183
 - main, creating a, 1172, 1173–1174, 1175–1180
 - menu bar, 1171, 1173–1174, 1175
 - mnemonics, 1171, 1181
 - popup, 1172, 1185
 - radio menu items, 1172, 1183–1185
 - standard menu, 1171
 - and submenus, 1172, 1174, 1179
 - and toolbars, 1171, 1173, 1189–1190
 - and tooltips, 1189
- Menu(s), Swing, 1069–1101
 - accelerator keys, 1069, 1078, 1079–1080, 1093
 - action command string, 1069–1070, 1078
 - action to manage multiple components of a, using an, 1069, 1089–1094
 - and check boxes, 1081, 1082–1083
 - classes, interaction of core, 1069–1070
 - demonstration program, 1095–1101
 - events, 1069–1070, 1073, 1081, 1082, 1084, 1085–1086
 - and images, 1080–1081
 - main, creating a, 1074–1078
 - menu bar, 1069, 1071–1072, 1074
 - mnemonics, 1069, 1073, 1078, 1079–1080, 1093
 - popup, 1069, 1070, 1083–1086
 - and radio buttons, 1081, 1082–1083
 - and submenus, 1070, 1072, 1077
 - and toolbars, 1069, 1070, 1087–1089
 - and tooltips, 1081
- MenuBar class
 - AWT, 799, 870, 871
 - JavaFX, 1172, 1173, 1175
- MenuDragMouseEvent, 1071
- MenuEvent, 1071
- MenuItem class
 - AWT, 799, 870–871, 872, 1081
 - JavaFX, 1172, 1173, 1174–1175, 1179, 1180, 1183
- MenuEvent, 1071
- MenuKeyEvent, 1071
- MenuListener, 1071
- Metadata, 280. *See also Annotation(s)*
- Method class, 282, 285, 286, 496, 1002, 1003, 1206
- Method reference(s), 381, 396–404
 - and the Collections Framework, 402
 - and generics, 401–404
 - to instance methods, 397–401
 - to static methods, 396–397
 - to a superclass version of a method, 401
- Method(s), 19, 110, 115–121
 - abstract. *See Abstract method(s)*
 - and annotations, 280, 299
- and autoboxing/unboxing, 275–276
- bridge, 374–375
- calling, 117, 118
- default interface, 16, 197, 207–211, 381, 383
- dispatch, dynamic, 178–181
- and the dot (.) operator, 111, 117, 118
- factory, 729
- final, 147, 184
- general form, 116
- generic, 338, 350, 356–359, 377
- getter, 1200
- hidden, using super to access, 170–171, 176
- inlining, 184
- interface. *See Interface methods*
- lookup, dynamic, 198
- native, 325–328, 491
- overloading, 129–134, 158–160, 177
- overriding. *See Overriding, method*
- and parameters, 116, 119–121
- passing an object to, 137–138
- recursive, 139–141
- reference. *See Method reference(s)*
- resolution, dynamic, 196, 198, 199, 204
- returning an object from, 138–139
- returning a value from, 118–119, 121
- scope defined by, 46–48
- setter, 1200
- static, 145–146, 211–212, 332–333, 396–397
- subclass responsibility, 182
- synchronized, 236, 247–249
- type inference and, 358, 372–373
- varargs. *See Varargs*
- variable-arity, 155
- MethodDescriptor class, 1202, 1205, 1206
- MethodHandle class, 496
- methodModifiers(), 1005
- MethodType class, 496
- MIME (Multipurpose Internet Mail Extensions), 1211, 1215
- min(), 444, 445, 450, 452, 479, 553, 556, 967, 971, 972
- minimumLayoutSize(), 856
- MIN_EXPONENT, 443
- MIN_NORMAL, 443
- MIN_PRIORITY, 246, 482
- MIN_RADIX, 455
- MIN_VALUE, 443, 447, 455
- mkdir(), 648
- mkdirs(), 648
- Model-Delegate component architecture, 1023–1024
- Model-View-Controller (MVC) component architecture, 1023
- Modifier class, 1003, 1005
 - “is” methods, table of, 1004
- Modulus operator (%), 61, 63
- Monitor, 236, 247, 249, 251
- Mouse events, handling, 785–788
- MouseAdapter class, 792, 793, 794, 1084, 1085, 1207
- mouseClicked(), 784, 792, 1084
- mouseDragged(), 784, 791, 792, 891
- mouseEntered(), 784, 1084
- MouseEvent class, 772, 774, 775, 778–779, 1084
- mouseExited(), 784, 1084

- MouseListener interface, 782, 784, 785–788, 792, 793, 1084, 1085–1086
 MouseMotionAdapter class, 791, 792
 MouseMotionListener interface, 771, 782, 784, 785–788, 791, 792, 793
 mouseMoved(), 784, 791, 891
 mousePressed(), 784, 793–794, 1084, 1086, 1207
 mouseReleased(), 784, 1084, 1086
 MouseWheelEvent class, 772, 779–780
 MouseWheelListener interface, 782, 784, 785, 792
 mouseWheelMoved(), 784
 Multi-core systems, 234–235, 261, 915, 916, 947–948, 952
 MultipleSelectionModel class, 1147, 1149
 multiplyExact(), 480
 Multitasking, 233
 preemptive, 235
 Multithreaded programming, 7, 11, 12, 233–261
 and context switching. *See* Context switching
 effectively using, 261
 and multi-core versus single-core systems, 234
 and spurious wakeup, 251
 and StringBuilder class, 439
 and synchronization. *See* Synchronization
 and threads. *See* Thread(s)
 versus the concurrency utilities, traditional, 915, 964
 and parallel programming, 948
 versus single-threaded system, 234
 MutableComboBoxModel, 1061
 MutableTreeNode interface, 1064
 MVC (Model-View-Controller) component architecture, 1023
-
- N**
- NAME, 760, 761
 Name-space collisions
 between instance variables and local variables, 125
 packages and, 187, 194, 334
 Naming class, 1006, 1007
 NaN, 443, 446
 nanoTime(), 468, 469, 955
 @Native built-in annotation, 290
 native modifier, 325
 Natural ordering, 494, 452
 naturalOrder(), 543
 Naughton, Patrick, 6
 NavigableMap interface, 531, 534, 539
 methods, table of, 535–536
 NavigableSet interface, 501, 507–508, 518, 519
 methods, table of, 507
 negateExact(), 480
 Negative numbers in Java, representation of, 66–67
 NEGATIVE_INFINITY, 443
 NegativeArraySizeException, 226, 557
 .NET Framework, 8
 NetBeans, 1112, 1212, 1213
 Networking, 727–745
 basics, 727–728
 classes and interfaces, list of, 728–729
 new, 52, 53, 113–114, 121, 123, 125, 139, 182, 222, 223
 autoboxing and, 275
 constructor reference and, 404, 408
 and enumerations, 264, 267
 and type inference, 372–373
 NEW, 260
 New I/O. *See* NIO
 newByteChannel(), 693, 696, 701, 702, 703, 704, 705, 706, 707, 708
 newCachedThreadPool(), 937
 newCondition(), 944
 newDirectoryStream(), 696, 714, 715–717
 newFilesystem(), 700
 newFixedThreadPool(), 937
 newInputStream(), 697, 709, 710–711
 Newline, inserting a, 612
 newOutputStream(), 697, 709, 711
 newScheduledThreadPool(), 937
 next(), 522, 523, 623, 863, 986, 987
 nextAfter(), 479
 nextBoolean(), 596, 623
 nextBytes(), 596
 nextDouble(), 205, 596, 623, 625, 628
 nextDown(), 479
 nextElement(), 562, 580, 665
 nextFloat(), 596, 623
 nextGaussian(), 596
 nextInt(), 596, 623, 628
 nextLong(), 596, 623
 nextToken(), 580
 nextUp(), 479
 nextX() Scanner methods, 621, 624, 625, 628
 table of, 623
 NIO, 641, 689–725
 and directories, 714–719
 packages, list of, 689
 pre-JDK 7 NIO versus new, 720
 reading a file using pre-JDK 7, 720–723
 for path and file system operations, using, 712–719
 and the stream API, 695
 for stream-based I/O, using, 700, 709–711
 writing to a file using pre-JDK 7, 723–725
 NIO and channel-based I/O
 copying a file using, 708–709
 reading a file using, 701–705
 writing to a file using, 705–708
 NIO.2, 689, 700, 712
 Node class, 1107, 1111, 1115, 1119, 1126, 1128, 1160, 1165, 1166, 1170, 1172, 1173, 1187
 Node(s), JavaFX, 1107, 1110, 1111, 1113, 1118, 1119
 disabling, 1170
 effects and transforms to alter the look of, using, 1164–1170
 hierarchy, 1107
 scrolling capabilities to, adding, 1157–1159
 text, 1170
 tree, 1160–1161, 1164
 noneMatch(), 990
 NORM_PRIORITY, 246, 482
 NoSuchElementException, 506, 508, 510, 534, 562, 624, 630
 NoSuchFieldException, 227

NoSuchMethodException, 227, 282
 NOT operator
 bitwise unary (`~`), 66, 67, 68–69
 Boolean logical unary (`!`), 75–76
 NotDirectoryException, 714
 notepad, 464, 467
`notify()`, 185, 186, 251, 253–255, 258–259, 471, 915,
 944, 964
`notifyAll()`, 185, 186, 251, 471
`notifyObservers()`, 598–599
 NotSerializableException, 687
`now()`, 1014–1015
 null, 34, 123
 alternative to using, 584
 Null statement, 90
 NullPointerException, 223, 226, 502, 504, 506, 508,
 510, 521, 531, 534, 557, 570, 631, 665
 using `Optional` to prevent a, 584, 586
`nullsFirst()`, 543
`nullsLast()`, 543
 Number class, 273, 347, 442
 NumberFormatException, 226, 273, 762
`numberOfLeadingZeros()`, 450, 452
`numberOfTrailingZeros()`, 451, 453
 Numbers, formatting, 609–610, 612–618

0

Oak, 6
 Object class, 185–186, 338, 340, 373, 471–473
 as a data type, problems with using the, 342–344
 Object class methods
 and functional interfaces, 382
 table of, 185, 471
 Object reference variables
 and abstract classes, 182, 184
 and argument passing, 136, 137–138
 assigning, 115
 declaring, 113
 and cloning, 471–472
 and dynamic method dispatch, 178–181
 to superclass reference variable, assigning
 subclass, 166, 170
 OBJECT tag, 320, 748, 761
 Object-oriented programming (OOP), 5, 6, 17–23, 109
 model in Java, 11
 Object(s), 19, 109, 114
 bitwise copy (`clone`) of, 471
 creating/declaring, 111, 113–114
 initialization with a constructor, 121, 123–124
 to a method, passing, 137–138
 monitor, implicit, 236, 249
 as parameters, 134–136
 returning, 138–139
 serialization of. *See* `Serialization`
 type at run time, determining, 322–324
 Object.notify(). *See* `notify()`
 Object.wait(). *See* `wait()`
 ObjectInput interface, 685
 methods defined by, table of, 685
 ObjectInputStream class, 303, 685
 methods defined by, table of, 686

ObjectOutput interface, 683, 684
 methods defined by, table of, 683
 ObjectOutputStream class, 303, 684
 methods defined by, table of, 684
 Objects class, 635
 Observable class, 598–601
 methods, table of, 598
`observableArrayList()`, 1146, 1149, 1151
 ObservableList, 1113, 1114, 1146, 1149, 1150, 1151, 1173
 ObservableValue, 1139
 Observer interface, 598–601
 Octals, 41
 as character values, 43
`of()`, 521, 522, 584, 585, 990
`offer()`, 508, 520
`offerFirst()`, 509, 510, 515
`offerLast()`, 509, 510, 515
`offsetByCodePoints()`, 431, 438
`ofLocalizedDate()`, 1015
`ofLocalizedDateTime()`, 1015
`ofLocalizedTime()`, 1015
`ofNullable()`, 585, 586
`ofPattern()`, 1016–1017
 pattern letters, 1016–1017
`onAdvance()`, 933–934, 936
`open()`, 693
`openConnection()`, 736, 738–739
 OpenOption interface, 695
 Operator(s)
 arithmetic, 61–66
 assignment. *See* Assignment operator(s)
 bitwise, 66–74
 Boolean logical, 75–77
 conditional-and, 77
 conditional-or, 77
 diamond (`<>`), 372–373
 parentheses and, 41, 79
 precedence, table of, 78
 relational, 28, 40, 41, 74–75
 ternary if-then-else (`?:`), 75, 77–78
 Optional class, 584–586, 971, 972, 973
 methods, table of, 584–585
 OptionalDouble class, 584, 586
 OptionalInt class, 584, 586
 OptionalLong class, 584, 586
 OR operator
 bitwise (`|`), 66, 67, 68–69
 bitwise exclusive (`^`), 66, 67, 68–69
 Boolean logical (`|`), 75–76
 Boolean logical exclusive (`^`), 75–76
 OR operator, short-circuit (`||`) Boolean logical,
 75, 76–77
 Oracle, 14, 1212
 Ordinal value, enumeration constant's, 269
`ordinal()`, 269, 270, 492
`orElse()`, 585
`out` output stream, 26, 34, 304, 305, 308, 309, 464, 467,
 620, 665, 666, 680
`out()`, 606, 608
 OutputStream class, 302, 303, 308, 650, 651, 654, 659,
 661, 665, 667, 679, 684, 711, 1220
 methods, table of, 652
 OutputStreamWriter class, 304

- Overloading methods, 129–134, 158–160, 177, 375–376
 @Override, built-in annotation, 290, 292
 Overriding, method, 175–181
 - and abstract classes, 181–184
 - and bridge methods, 374–375
 - and dynamic method dispatch, 178–181
 - final to prevent, using, 184
 - in a generic class, 371–372
 - and run-time polymorphism, 178, 179, 181
- P**
- Package(s), 142, 187–196, 212
 - access to classes contained in, 190–194, 195
 - built-in standard Java classes and, 194
 - core Java API, table of, 991–993
 - the default, 188, 194
 - defining, 188
 - finding, 188–189
 - importing, 194–196
 - Swing, 1024
 - version data, obtaining, 489
- Package class, 286, 489–490
 - methods, table of, 489–490
- package statement, 188, 194
- Paint class, 1121
- Paint mode, setting, 818–819
- paint(), 319, 751–752, 753, 754, 755–756, 757, 759, 786, 805, 811, 887, 891, 895, 1033, 1036, 1037
 - lightweight AWT components and overriding, 882–883
- Paintable area, computing, 1037
- paintBorder(), 1036
- paintChildren(), 1036
- paintComponent(), 1036, 1037, 1040
- Painting in Swing, 1036–1040
- Panel class, 749, 799, 800, 801, 863
- Panes, Swing container, 1025. *See also* Content pane
- Parallel processing, 16, 381, 526, 528
 - of a stream API stream, 965, 968, 969, 975–977, 984, 986, 987, 989
- Parallel programming. *See* Programming, parallel
- parallel(), 966, 975
- parallelPrefix(), 560
- parallelSetAll(), 560
- parallelSort(), 559
- parallelStream(), 502, 504, 969, 975, 976
- PARAM NAME and VALUE, 760, 761
- Parameter(s), 25, 116, 119–121
 - applets and, 761–764
 - and constructors, 123–124
 - final, 147
 - and lambda expressions, 382–383, 385–387, 395
 - objects as, 134–136
 - and overloaded constructors, 134
 - and overloaded methods, 129, 177
 - and the scope of a method, 46
 - servlet, reading, 1220–1222
 - type. *See* Type parameter(s)
 - variable-length (varargs), 157, 521
- Parameterized types, 338, 340
- parameterModifiers(), 1005
- Parent class, 1007, 1111, 1115
- parse(), 1017–1018
- parseBoolean(), 460
- parseByte(), 448, 454
- parseDouble(), 445
- parseFloat(), 444
- parseInt(), 451, 454
- parseLong(), 453, 454
- parseShort(), 449, 454
- parseUnsignedInt(), 451
- parseUnsignedLong(), 453
- Parsing, definition of, 579
- Pascal, 4
- PasswordField class, 1156
- Passwords, reading, 680
- Path interface, 642, 645, 694–695, 700, 701, 712, 714, 720
 - converting a File object into an instance of the, 645, 695, 712
 - instance for stream-based I/O, using a, 709–711
 - methods, table of a sampling of, 694–695
 - obtaining an instance of the, 698, 700, 701, 702, 703–704, 707
- Paths class, 698, 700
- Pattern class, 993–994, 997, 1000, 1001
- Pattern matching, regular expressions, 995–1001
- PatternSyntaxException, 995
- Payne, Jonathan, 6
- peek(), 508, 567
- peekFirst(), 509, 515
- peekLast(), 509, 515
- Peers, native, 883, 1021–1022
- Period class, 1018
- Persistence (Java Beans), 1203
- Phaser class, 916, 917, 930–936
 - compatibility with fork/join, 963
- PI (Math constant), 477
- PIPE, 465
- Pipeline for actions on stream API streams, 16, 381, 968, 980
- PipedInputStream class, 303
- PipedOutputStream class, 303
- PipedReader class, 304
- PipedWriter class, 304
- PixelGrabber class, 897–899
- Platform class, 1180
- Platform.exit(), 1180
- play(), 750, 767
- Pluggable look and feel (PLAF), 1022–1023, 1024
- PNG file format, 886, 887
- Point class, 778, 779, 799
- Pointers, 59, 113
- poll(), 508, 520
- pollFirst(), 507, 509, 515
- Polling, 234, 251
- pollLast(), 507, 509, 515
- Polygon class, 799, 813
- Polymorphism, 5, 18, 21–23
 - and dynamic method dispatch, 178–181, 182
 - and interfaces, 196, 199, 204
 - and overloaded methods, 129, 131, 132
- pop(), 509, 510, 567

- PopupControl class, 1173, 1185
 PopupMenu class, 799, 876
 PopupMenuItem, 1071
 Port, 727–728, 735
 Portability problem, 6–7, 8, 9, 10, 12, 16
 and data types, 36
 and native methods, 328
 and thread context switching, 235
 Pos enumeration, 1119
 POSITIVE_INFINITY, 443
 PosixFileAttributes class, 699, 714
 PosixFileAttributeView interface, 699
 postVisitDirectory(), 718
 pow(), 331–333, 478
 Predicate<T> predefined functional interface, 409, 503, 638, 972
 preferredLayoutSize(), 856
 previous(), 523, 863
 preVisitDirectory(), 718
 PrimitiveIterator interface, 499, 986
 PrimitiveIterator.OfDouble interface, 499, 986
 PrimitiveIterator.OfInt interface, 499, 986
 PrimitiveIterator.OfLong interface, 499, 986
 print(), 27, 34, 308, 309, 418, 666, 680, 1220
 printf()
 function, C/C++, 605, 666
 method, Java, 155, 620, 666–667, 679, 680, 681
 println(), 26, 27, 34, 186, 308, 309, 418–419, 608, 665, 666, 679, 680, 1220
 and applets, 748, 767
 and Boolean output, 41
 and String objects, 59
 printStackTrace(), 228
 PrintStream class, 303, 305, 308, 620, 665–667
 PrintWriter class, 304, 308–309, 620, 679–680, 1215
 PriorityBlockingQueue class, 943
 PriorityQueue class, 511, 519–520
 private access modifier, 25, 142–144, 190–191
 and inheritance, 163–164
 Process class, 460–461, 464, 465
 methods, table of, 461
 Process, definition of, 233, 460
 Process-based versus thread-based multitasking, 233
 ProcessBuilder class, 460, 465–467
 methods, table of, 465–466
 ProcessBuilder.Redirect class, 465
 ProcessBuilder.Redirect.Type enumeration, 465
 Program, creating a windowed, 809–810
 Programming
 multithreaded. *See* Multithreaded programming
 object-oriented. *See* Object-oriented programming
 process-oriented, 17, 18, 22
 structured, 4, 5
 Programming, parallel, 15, 235, 916, 917, 947–948, 975
 and specifying the level of parallelism, 950, 955–958, 963
 Project Coin, 14
 Properties class, 498–499, 561, 572–577
 methods, table of, 573–574
 Properties, environment, 470
 Property, Java Bean, 1208
 bound and constrained, 1203, 1206
 design patterns for, 1200–1201, 1203
 PropertyChangeEvent, 1203
 PropertyChangeListener interface, 1203, 1204
 PropertyDescriptor class, 1202, 1205, 1206, 1208
 PropertyPermission class, 635
 PropertyResourceBundle class, 633
 PropertyVetoException, 1203
 protected access modifier, 126, 142, 190–191
 Protocols, overview of networking, 727–728
 Pseudorandom numbers, 596
 public access modifier, 25, 142–144, 190–191
 Push buttons, AWT, 751, 836–840
 action command string of, 836, 838, 839
 Push buttons, JavaFX, 1115–1119
 adding an image to, 1130–1133
 Push buttons, Swing, 1030–1033, 1045–1047
 action command string of, 1045
 push(), 510, 567
 Pushback, 662
 PushbackInputStream, 303, 659, 662–663
 PushbackReader class, 304, 678–679
 put(), 531, 533, 537, 539, 541, 568, 569, 570
 and buffers, 691, 692, 706–707, 724
 putAll(), 533, 541
 PutField inner class, 684
 putValue(), 1090–1091, 1092
-
- ## Q
-
- Query string, 1228
 Queue interface, 501, 508, 515, 519, 520
 methods, table of, 508
 quietlyInvoke(), 962
 quietlyJoin(), 962
-
- ## R
-
- Race condition, 248–249
 Radio buttons, 842
 JavaFX, 1135–1138, 1139–1142
 Swing, 1051–1053
 and Swing menus, 1082–1083
 RadioButton class, 1135, 1136, 1142
 RadioMenuItem class, 1172, 1183
 Radix, 447
 radix(), 630
 Random class, 205, 596–598
 methods, table of core, 596
 random(), 480
 RandomAccess interface, 501, 530
 RandomAccessFile class, 669–670, 692, 724
 range(), 521, 522
 Raw types, 362–364
 READ, 465
 read(), 93, 303, 304, 305–306, 310–311, 495, 651, 660, 662, 671, 678, 685, 686, 693, 701, 702, 703, 710, 721
 Readable interface, 495, 620, 626, 670
 ReadableByteChannel interface, 620
 readAttributes(), 697, 699, 712–714
 readBoolean(), 668, 686
 readDouble(), 668, 686
 Reader class, 303–304, 305, 650, 670, 672, 674, 688
 methods defined by, table of, 671

readExternal(), 683
readInt(), 668, 686
readLine(), 306–308, 454, 680, 681, 751, 1220
readObject(), 685, 686
readPassword(), 680, 681
ReadWriteLock interface, 946
Real numbers, 38
rebind(), 1006
receive(), 743
Recursion, 139–141

- and the Fork/Join Framework divide-and-conquer strategy, 951–952

RecursiveAction class, 917, 948, 949–950, 951, 952, 954, 958
RecursiveTask class, 917, 948, 950, 951

- example program using, 958–960

Redirect class, 465
reduce(), 968, 973–977, 978–979
Reduction operations, 973–975

- mutable, 985

ReentrantLock, 944
ReentrantReadWriteLock, 946
Reflection, 281, 496, 991, 1001–1005

- and annotations, 281–286

ReflectiveOperationException, 227
Region class, 1107, 1115
regionMatches(), 421–422
register(), 930
Regular expressions, 432, 621, 628, 991, 993–1001

- syntax, 995
 - wildcards and quantifiers, 993, 995, 997–999

reinitialize(), 961
Relational operators, 28, 40, 41, 74–75
Relative index, 618–619
release(), 918–921
remainderUnsigned(), 451, 453
Remote interface, 1006
Remote method invocation (RMI), 12, 682, 991, 1005–1009
RemoteException, 1006
remove(), 502, 503, 505, 508, 516, 521, 522, 523, 533, 568, 569, 570, 834, 1029, 1071, 1072, 1114, 1166, 1173, 1174
removeActionListener(), 1032
removeAll(), 502, 503, 834
removeAttribute(), 1225, 1232
removeEldestEntry(), 541
removeElement(), 563, 564
removeElementAt(), 563, 564
removeFirst(), 510, 515
removeIf(), 502, 503
removeKeyListener(), 771
removeLast(), 510, 515
removeTListener(), 1202
removeTypeListener(), 771
renameTo(), 644
repaint(), 756–759, 803, 1037, 1119, 1208
@Repeatable annotation, 290, 297, 298
replace(), 427, 437, 533
replaceAll(), 431, 504, 505, 533, 995, 999–1000
replaceFirst(), 431
replaceRange(), 854

ReplicateScaleFilter class, 899
reset(), 630, 651, 652, 657, 660–661, 663, 671, 676, 691
resolve(), 694, 695
Resource bundles, 630–634
 ResourceBundle class, 630–633

- methods, table of, 631–632

 ResourceBundle.Control class, 631
resume(), 13, 257, 482, 488
retainAll(), 503
@Retention built-in annotation, 281, 290
RetentionPolicy enumeration, 281, 496
return statement, 108, 116

- in a lambda expression, 388–389

reverse(), 436, 451, 453, 553
reverseBytes(), 449, 451, 453
reversed(), 542, 545–546
reverseOrder()

- collection algorithm, 553, 555–556
- Comparator method, 543

rewind(), 691, 703, 706, 707, 721, 724
RGB (red-green-blue) color model, 816–817

- default, 895

RGBImageFilter class, 899, 902

- example program demonstrating the, 902–912

RGBCtoHSB(), 816
Richards, Martin, 4
rint(), 479
Ritchie, Dennis, 4
rmi protocol, 1007
RMI (Remote Method Invocation), 12, 682, 991, 1005–1009
rmic compiler, 1008
rmiregistry (RMI registry), 1008, 1009
Rotate class, 1166–1167

- program demonstrating, 1167–1170

rotateLeft(), 451, 453
rotateRight(), 451, 453
round(), 479
Run-time

- system, Java, 9. *See also* Java Virtual Machine (JVM)
- type information, 13, 322, 368–370, 376

run(), 236, 239, 240, 382, 481, 483, 602, 603, 962, 1030

- overriding, 241, 242, 602
 - using a flag variable with, 257–259, 759

RUNNABLE, 260
Runnable interface, 236, 238, 382, 481, 602, 758–759, 915, 961, 962, 1030

- implementing the, 239–240, 242

Runtime class, 460, 461–464, 958

- executing other programs and, 464
- memory management and, 462–464
- methods, table of some, 461–462

RUNTIME retention policy, 281, 282, 285
RuntimeException class, 214–215, 223, 226, 230
RuntimePermission class, 490

S

@SafeVarargs built-in annotation, 290, 292
SAM (Single Abstract Method) type, 382
save(), 573
scalb(), 478

Scale class, 1166, 1167
 program demonstrating, 1167–1170

Scanner, 579

Scanner class, 620–630
 closing an instance of the, 626
 constructors, 620, 621
 delimiters, 621, 628–629
 demonstration programs, 624–628
 hasNextX() methods, table of, 622
 how to use, 620–621, 623–624
 methods, miscellaneous, 629–630
 nextX() methods, table of, 623

Scene class, 1106, 1107, 1110, 1111

schedule(), 602, 603

ScheduledExecutorService interface, 937

ScheduledThreadPoolExecutor class, 917, 939

Scientific notation, 42, 607, 609–610

Scopes in Java, 45–48

Scroll bars, 773, 782, 849–851, 1056, 1149–1150, 1157–1159, 1160

Scroll pane, 1056–1057, 1157–1159

Scrollbar class, 799, 849

ScrollPane class, 1157–1159

search(), 567

Security manager, 310, 467, 490, 660, 1212

Security problem, 8, 9–10, 16
 and native methods, 328
 and servlets, 1212

SecurityException, 226, 310, 461, 467, 649, 666, 714, 717

SecurityManager class, 490

seek(), 670

SeekableByteChannel interface, 693, 701, 704, 705

select(), 844, 847, 852, 854

selectedItemProperty(), 1147, 1149, 1160

selectedToggleProperty(), 1139

Selection statements, 81–89

SelectionMode, 1150

SelectionModel class, 1147

Selectors, 693

Semaphore, 915, 916, 918–923
 and setting initial synchronization state, 923

Semaphore class, 916, 917, 918

send(), 743

Separable Model architecture, 1023

Separator class, 1142

SeparatorMenuItem class, 1172, 1174

Separators, 33

SequenceInputStream class, 303, 663–665

sequential(), 966, 977

Serializable interface, 682–683, 687, 962, 1203

Serialization, 682–688
 example program, 686–688
 and Java Beans, 1203
 and static variables, 683
 and transient variables, 683, 687

Server, 727

ServerSocket class, 692, 731, 741–742

service(), 1212, 1215, 1217, 1221, 1227

ServiceLoader class, 635

Servlet interface, 1216, 1217, 1220
 methods, table of, 1217

Servlet(s), 10, 16, 1211–1233
 advantages of, 1212
 API, 1216
 development options, 1212–1214
 example program for a simple, 1214–1216
 life cycle of, 1212
 parameters, reading, 1220–1222
 and portability, 10
 and security, 1212
 and session tracking, 1232–1233
 using Tomcat to develop, 1212, 1213–1216

ServletConfig interface, 1216, 1218, 1220

ServletContext interface, 1216, 1218
 methods, table of various, 1218

ServletException class, 1217, 1220

ServletInputStream class, 1217, 1220

ServletOutputStream class, 1217, 1220

ServletRequest interface, 1215, 1216, 1218, 1220
 methods, table of various, 1219

ServletResponse interface, 1215, 1216, 1218
 methods, table of various, 1219

Session tracking, HTTP, 1232–1233

Set interface, 501, 504, 506, 516, 521, 531, 536
 from a stream API stream, obtaining a, 982–984

Set-view, obtaining, 538–539, 571–572

set(), 504, 505, 516, 523, 582, 589, 946

setAccelerator(), 1079, 1181

setActionCommand(), 839, 872, 1045, 1052

setAlignment(), 835, 1119

setAll(), 560

setAllowIndeterminate(), 1145

setAngle(), 1166–1167

setAttribute(), 1218, 1225, 1232

setBackground(), 754–755, 816

setBlockIncrement(), 849

setBorder(), 1040

setBounds(), 801, 856

setChanged(), 598–599

setCharAt(), 434

setColor(), 817

setConstraints(), 866

setContent(), 1157

setContentDisplay(), 1129–1130, 1132–1133, 1190

contentType(), 1215, 1219

setContextMenu(), 1186

setDefault(), 593, 595

setDefaultCloseOperation(), 1028

setDisable(), 1170, 1175

setDisabledIcon(), 1045, 1081

setEchoChar(), 852

setEditable(), 852, 854, 1153–1154

setEffect(), 1165

setEnabled(), 871, 1073, 1091

setFill(), 1120–1121

setFont(), 822

setForeground(), 754–755, 816

setForkJoinTaskTag(), 962

setGraphic(), 1130, 1174, 1183

setHorizontalTextPosition(), 1081

setHvalue(), 1157–1158

setIcon(), 1042, 1080–1081

SetIntField(), 327

setJMenuBar(), 1072, 1074
setLabel(), 836, 840, 871
setLastModified(), 645
setLayout(), 856, 1029
setLength(), 433–434, 670, 744
setLevel(), 1165
setLocation(), 801
setMaxAge(), 1226, 1232
setMnemonic(), 1078
setMnemonicParsing(), 1181
setMultipleMode(), 882
setN() setter method design pattern, 1200, 1201
setName(), 237, 238, 484
setOnAction(), 1116, 1118, 1119, 1175
setOnContextMenuRequested(), 1187–1188
setOrientation(), 1189
setPaintMode(), 818
setPannable(), 1157
setPivotX(), 1166–1167
setPivotY(), 1166–1167
setPrefColumnCount(), 1154
setPreferredSize(), 801, 850
setPrefHeight(), 1147
setPrefSize(), 1147
setPrefViewportHeight(), 1157
setPrefViewportWidth(), 1157
setPrefWidth(), 1142, 1147
setPressedIcon(), 1045
setPriority(), 246, 484
setPromptText(), 1154
setReadOnly(), 645
setRolloverIcon(), 1045
setRotate(), 1166
setScaleX(), 1166
setScaleY(), 1166
setScene(), 1111
setSelected(), 1136, 1139, 1183
setSelectedCheckbox(), 842
setSelectedIcon(), 1045
setSelectionMode(), 1058, 1150
setSize(), 564, 801, 802, 803, 1028
setStackTrace(), 228
setState(), 840, 871
setStroke(), 1120–1121
setText(), 835, 852, 854, 1042, 1045, 1118, 1154, 1174
setTitle(), 802, 1110
setToggleGroup(), 1136
setTooltip(), 1170, 1189
setToolTipText(), 1081
setTranslateX(), 1166
setTranslateY(), 1166
setUnitIncrement(), 849
setValue(), 537, 849, 1090, 1091, 1226, 1152
setValues(), 849
setVvalue(), 1157–1158
setVisible(), 802, 803, 1029
setX(), 1167
setXORMode(), 818–819
setY, 1167
Shear class, 1166
Sheridan, Mike, 6
Shift operators, bitwise, 66, 69–73
Short class, 272, 273, 447, 454
 methods defined by, table of, 449
short data type, 35, 36, 37, 41
 and automatic type conversion, 48
 and automatic type promotion, 50, 69
ShortBuffer class, 691
shortValue(), 273, 442, 444, 446, 448, 449, 451, 453
show(), 863, 1084, 1085, 1086, 1111, 1188
showDocument(), 764, 765–766
showStatus(), 750, 759, 766, 792, 793–794
shuffle(), 553, 555–556
shutdown(), 937, 939, 951, 963
shutdownNow(), 963
Sign extension, 72–73
signal(), 944
signum(), 451, 453, 480
SimpleBeanInfo class, 1203, 1208, 1209
SimpleDateFormat class, 596, 1011–1013, 1017
 formatting string symbols, table of, 1012
SimpleFileVisitor class, 718, 719
SimpleTimeZone class, 594
sin(), 38, 477
SingleSelectionModel, 1053
sinh(), 477
SIZE, 443, 447
size(), 503, 516, 533, 564, 568, 569, 570, 582, 697, 698, 705, 1173, 1174
skip(), 630, 651, 652–654, 660, 671, 685
SKIP_SIBLINGS, 718
SKIP_SUBTREE, 718
sleep(), 236, 237–238, 243, 484, 943
slice(), 691
Slider box, 849
Socket class, 692, 731–734, 741, 742
Socket(s)
 datagram, 742–743, 744–745
 overview, 727
 TCP/IP client, 731–734
 TCP/IP server, 731, 741–742
SocketAddress class, 742–743
SocketChannel class, 692, 693
SocketException, 743
sort(), 504, 505, 554, 558–559
sorted(), 968, 972
SortedMap interface, 531, 534
 methods, table of, 534
SortedSet interface, 501, 506
 methods, table of, 506
Source code file, naming a, 23
SOURCE retention policy, 281
split(), 431, 432, 1000–1001
Spliterator, 499, 504, 526–529
 and arrays, 559
 characteristics, 528
Spliterator interface, 495, 499, 501, 523, 526–529
 methods declared by, table of, 527
 and streamAPI streams, 986, 987–989
spliterator(), 494–495, 503, 504, 559, 966
Spliterator.OfDouble interface, 529
Spliterator.OfInt interface, 529
Spliterator.OfLong interface, 529
Spliterator.OfPrimitive interface, 529
sqrt(), 38, 45, 331–333, 478
Stack
 definition of, 21, 127
 ways to implement a, 201

- Stack class, 498–499, 511, 561, 566–568
 - methods, table of, 567
- Stack frame, 491
- Stack trace, 215–216, 222, 491
- StackTraceElement class, 228, 491
 - methods, table of, 491
- Stage class, 1106, 1107, 1110, 1111
- StampedLock interface, 946
- StandardCopyOption values, partial list of, 709
- StandardOpenOption class, 695, 710, 711
 - enumeration, table of values for the, 697
- StandardOpenOption.CREATE, 697, 705, 711
- StandardOpenOption.READ, 697, 708, 710
- StandardOpenOption.TRUNCATE_EXISTING, 697, 711
- StandardOpenOption.WRITE, 697, 705, 711
- Standard Template Library (STL), 499
- start(), 236, 239, 240, 241, 460, 466, 467, 484, 750, 751, 753, 755, 803, 994, 997, 1033, 1035, 1107, 1108, 1110, 1111
- startsWith(), 422, 695
- State enumeration, 259
- Statements, 26
 - null, 90
- Statements, control, 28, 40
 - iteration, 81, 89–102
 - jump, 81, 102–108
 - selection, 81–89
- static, 25, 145–146, 149, 325, 331, 332–333
 - member restrictions, 377
- Static import, 14, 331–334
- stop(), 13, 257, 482, 750, 751, 753, 756, 759, 767, 803, 1033, 1107, 1108, 1110, 1111, 1180
- store(), 573, 576–577
- Stream API, 16, 965–990
 - and collections, 577, 965
 - interfaces, 966–968
 - and lambda expressions, 965
 - and NIO, 695
- Stream interface, 504, 559, 676, 695, 967–968, 989, 990
 - methods, table of some, 967–968
- Stream, intermediate operations on a stream API, 968
 - to create a pipeline of actions, 968, 980
 - lazy behavior of, 968
 - stateless versus stateful, 968
- Stream, stream API
 - collection from a, obtaining a, 982–985
 - definition of a, 965–966
 - iterators and a, 986–989
 - mapping a, 978–982
 - obtaining a, 969
 - operations on a, terminal versus
 - intermediate, 968
 - ordered versus unordered, 977
 - parallel processing of a, 965, 968, 969, 975–977, 984, 986, 987, 989
 - parallel, using a, 975–977, 989
 - reduction operations, 973–975
- stream(), 503, 504, 559, 582, 969, 971, 976, 981
- Stream(s), byte, 302–303, 305, 309, 650, 651–670
 - classes in java.io, table of, 303
- Stream(s), character, 302, 303–304, 305, 309, 650, 670–680
 - classes in java.io, table of, 304
- Stream(s), I/O
 - benefits, 688
 - buffered, 659–663, 676–678
 - classes, top-level, 650
 - closing, 649–650
 - concatenating input to, 663–665
 - definition of, 302, 641
 - filtered, 659, 688
 - flushing, 648
 - and NIO, 709–711
 - predefined, 304–305
- strictfp, 324
- StrictMath class, 481
- String class, 25, 58–59, 152–154, 413, 493, 620, 761, 994
 - constructors, 414–416
 - methods, table of some, 431–432
- String(s)
 - arrays of, 58, 154
 - changing case of characters in, 429–430, 456, 457
 - comparison, 153, 420–424
 - concatenating, 152–153, 417–418, 427, 435
 - constants, 58, 152
 - converting data into a, 418–419, 428–429
 - creating, 152, 414–416
 - extracting characters from, 419–420
 - formatted, creating a, 607–608
 - formatting a, 609, 614
 - immutability of, 152, 413, 426, 432
 - joining, 430–431
 - length, obtaining, 153, 416
 - literals, 43–44, 416
 - modifiable, creating and working with, 432–439
 - modifying, 426–428
 - numbers to and from, converting, 454–455
 - as objects, 44, 58–59, 152, 413
 - parsing a formatted input, 579
 - reading, 306–308
 - searching, 424–426
- StringBuffer class, 152, 413, 415, 426, 432–439, 493
 - methods, table of some, 439
- StringBufferInputStream deprecated class, 642
- StringBuilder class, 152, 413, 415–416, 426, 439, 493, 606
 - and synchronization, 439
- StringIndexOutOfBoundsException exception, 226
- StringJoiner class, 635
- StringReader class, 304
- StringTokenizer class, 579–580, 832
 - methods, table of, 580
- stringWidth(), 826
- StringWriter class, 304
- strokeLine(), 1120
- strokeOval(), 1120
- strokeRect(), 1120
- strokeText(), 1120
- Stroustrup, Bjarne, 6
- Stubs (RMI), 1007–1008
- Subclass, 20, 161–164, 179
- subList(), 504, 505
- subMap(), 534, 536
- submit(), 940
- subSequence(), 432, 438, 493
- subSet(), 506, 507, 519

- substring(), 426–427, 437
 subtractExact(), 480
 sum(), 444, 446, 451, 453
 Sun Microsystems, 6, 14
 super, 145, 167
 - and bounded wildcard arguments, 356
 - and interface default methods, 211
 - and method references, 401
 - and methods or instance variables, 170–171, 176
 super(), 336
 - and superclass constructors, 167–170, 174
 Superclass, 20, 161–164, 179, 187
 - abstract, 181–184
 Supplemental character, definition of, 458
 Supplier<T> predefined functional interface, 409, 638, 985
 @SuppressWarnings built-in annotation, 290, 292
 suspend(), 13, 257, 482, 488
 Swing, 13, 16, 301, 319, 747, 797, 833, 1021–1068, 1105
 - applet, example of a simple, 1033–1035
 - application, example of a simple, 1026–1030
 - and the AWT, 797, 1021–1022
 - component classes, list of, 1024–1025
 - components. *See* Components, Swing
 - event handling, 1030–1033
 - history of, 1021–1022
 - and JavaFX, 1106, 1119
 - menus. *See* Menu(s), Swing
 - and MVC architecture, 1023
 - packages, list of, 1026
 - and painting, 1033, 1036–1040
 - threading issues, 1029–1030, 1033*Swing: A Beginner's Guide* (Schildt), 1021
 SwingConstants interface, 1042
 SwingUtilities class, 1030
 switch statement, 84–89
 - and auto-unboxing, 277
 - nested, 88
 - using enumeration constants to control a, 84, 264–265
 - using a String to control a, 15, 84–85, 87–88
 - versus the if statement, 88–89
 Synchronization, 12, 235–236, 247–250
 - and atomic operations, 946–947
 - and collections, 510, 550
 - and deadlock, 255–257
 - and interprocess communication, 251–257
 - objects, using, 917–936
 - race condition and, 248–249
 - and StringBuilder class, 439
 - via synchronized block, 249–250, 550
 - via synchronized method, 236, 247–249
 - versus concurrency utilities, traditional, 915, 964
 synchronized modifier, 247, 915, 943, 964
 - used with a method, 247–249
 - used with an object, 249–250
 synchronizedList(), 550, 554
 synchronizedSet(), 550, 554
 Synchronizers, 915, 916–917
 SynchronousQueue class, 943
 System class, 26, 34, 304, 467–470
 - methods, table of, 467–468
 System.console(), 467, 680
 System.err standard error stream, 304, 305, 467
 System.exit(), 1078, 1180
 System.getProperties(), 468, 572
 System.getProperty(), 468, 470
 System.in standard input stream, 304, 305, 464, 467, 620, 680
 System.in.read(), 93
 System.nanoTime(), 468, 469, 955
 System.out standard output stream, 26, 34, 304, 305, 308, 309, 464, 467, 620, 665, 666, 680
 - and static import, 333, 334
 System.out.println() and applets, 748, 767

T

-
- Tabbed panes, 1053–1055
 Table, Swing, 1066–1068
 TableColumnModel, 1067
 TableModel, 1067
 TableModelEvent class, 1067
 tailMap(), 534, 536
 tailSet(), 506, 507
 tan(), 477
 tanh(), 477
 @Target built-in annotation, 290, 291
 TCP/IP, 12, 728
 - client sockets, 731–734
 - disadvantages of, 742
 - server sockets, 731, 741–742
 - See also* Transmission Control Protocol (TCP)
 TERMINATE, 718
 TERMINATED, 260
 Ternary if-then-else operator (?::), 75, 77–78
 test(), 638, 972
 Text area, 780, 854–855
 Text class, 1170
 Text components as an event source, 782
 Text fields, 780
 - AWT, 852–854
 - Swing, 1043–1044
 Text formatting using java.text classes, 991, 1009–1013
 Text output using font metrics, managing, 825–832
 TextArea class
 - AWT, 800, 854–855
 - JavaFX, 1156
 TextComponent class, 800, 852, 854
 TextEvent class, 772, 780, 854
 TextField class
 - AWT, 800, 852
 - JavaFX, 1154–1156
 TextInputControl, 1154
 TextListener interface, 782, 784
 textValueChanged(), 784
 thenComparing(), 543–544, 548–550
 thenComparingDouble(), 544
 thenComparingInt(), 544
 thenComparingLong(), 544
 this, 124–125, 145
 - and lambda expressions, 395
 - and type annotations, 292, 293, 296
 this(), 334–336
 Thompson, Ken, 4

- Thread class, 13, 236, 237, 481, 482–484, 602, 759, 915
 constructors, 239, 242, 482
 extending, 241–242
 methods, table of, 482–484
- Thread(s)
 creating, 238–243
 daemon, 602, 951, 961
 and deadlock, 255–257, 482, 1030
 definition of, 233
 executors to manage, using, 917, 937–942
 group, 238, 484–488
 JavaFX, 1112
 local variables, 488
 main, 236–238, 240, 242, 243, 244
 messaging, 236, 251–255
 pool, 937–939, 949, 950, 951, 954–955, 958, 963
 priorities, 235, 246, 482
 resuming, 257–261, 486–488
 and spurious wakeup, 251
 states of, possible, 235, 259–261
 stopping, 257
 suspending, 236, 237–238, 257–261, 486–488
 and Swing, event dispatching, 1029–1030, 1033, 1034, 1035
 synchronization. *See* Synchronization
- Thread.UncaughtExceptionHandler interface, 495
- ThreadGroup class, 481, 484–488, 495
 methods, table of, 484–485
- ThreadLocal class, 488
- ThreadPoolExecutor class, 917, 937
- throw, 213, 222–223, 232
- Throwable class, 214–215, 218, 222, 223, 227, 230, 318, 379, 490, 491
 methods defined by, table of, 228
 obtaining an object of the, 222–223
- throws, 213, 223–224, 226
- Thumb, 849
- time, 588
- Time and date
 formatting, 610–612, 1009–1013, 1015–1017
 java.util classes that deal with, 586–596
 strings, parsing, 1017–1018
- Time and Date API, 1009, 1013–1018
 packages, list of, 1013
- timedJoin(), 943
- timedWait(), 943
- TIMED_WAITING, 260
- Timer class, 602–604
 methods, table of, 603
- TimerTask class, 602–604
 methods, table of, 602
- Timestamp, event, 773
- TimeUnit enumeration, 917, 923–924, 940, 942–943
- TimeZone class, 593, 594
 methods defined by, table of some, 593
- to(), 465
- toAbsolutePath(), 695, 712
- toArray(), 504–504, 513–514, 968
- toBinaryString(), 451, 453, 455
- toCharArray(), 420
- toDays(), 943
- toDegrees(), 480, 481
- ToDoubleFunction functional interface, 636, 981
- toFile(), 695
- Toggle button, JavaFX, 1133–1135
 adding an image to a, 1133
- Toggle button, Swing, 1047–1049
- Toggle interface, 1133, 1135, 1142
- Toggle group, 1136, 1138
 handling change events in a, 1138–1139
 and RadioMenuItem, 1183
- ToggleButton class, 1133–1135
- ToggleButton class, 1136, 1138, 1142
- toHexString(), 444, 446, 451, 453, 455
- toHours(), 943
- toIntExact(), 480
- Tokens, 579, 620–621, 628
- toLanguageTag(), 595
- toList(), 982, 984
- toLocalDate(), 1015
- toLocalTime(), 1015
- toLowerCase(), 429–430, 457, 458
- Tomcat, 1212, 1213–1216
- toMicros(), 942
- toMillis(), 942
- toMinutes(), 943
- toNanos(), 943
- toOctalString(), 451, 453, 455
- Toolbar class, 1173, 1189
- Toolbars, 1069, 1070, 1087–1089, 1171, 1173, 1189–1190
- Tooltip class, 1170
- Tooltips, 1081, 1087, 1088–1089, 1170, 1189–1190
- TooManyListenersException, 1202
- toPath(), 645, 695, 712
- toRadians(), 480, 481
- toSeconds(), 943
- toSet(), 982
- toString(), 185, 186, 218, 227, 228, 230, 273, 280, 286, 309, 418–419, 429, 444, 446, 448, 449, 451, 453, 454, 460, 471, 475, 484, 485, 490, 491, 492, 493, 513, 560, 564, 570, 582, 583, 585, 587, 604, 606, 608, 666, 680, 694, 695, 731, 772, 820, 826, 963, 1064
- totalMemory(), 462–463
- toUnsignedInt(), 448, 449
- toUnsignedLong(), 448, 449, 451
- toUnsignedString(), 451, 453
- toUpperCase(), 429–430, 457
- toZonedDateTime(), 592
- transient modifier, 322, 1203
- Transform class, 1166
- Transforms, 1166–1167
 program demonstrating, 1167–1170
- Translate class, 1166
- translatePoint(), 779
- Transmission Control Protocol (TCP)
 definition of, 727
 and stream-based I/O, 728
See also TCP/IP
- TreeExpansionEvent class, 1063
- TreeExpansionListener interface, 1063
- TreeItem class, 1160, 1161, 1164
- TreeMap class, 537 539–540, 542, 577
 example using a comparator, 549–550

- T**
 TreeModel, 1063
 TreeModelEvent class, 1063
 TreeModelListener interface, 1063
 TreeNode interface, 1064
 TreePath class, 1064
 Trees
 JavaFX, 1160–1164
 Swing, 1063–1066
 TreeSelectionEvent class, 1063
 TreeSelectionListener interface, 1063, 1064
 TreeSelectionModel, 1063
 TreeSet class, 511, 517, 518–519, 542, 577
 TreeView class, 1160–1163
 trim(), 428
 trimToSize(), 438, 513, 564
 true, 34, 40, 41, 43, 75, 76
 TRUE, 458
 True and false in Java, 43, 75
 Truncation, 49
 try block(s), 213, 214, 216–222, 224–225, 232
 nested, 220–222
 try-with-resources statement, 15, 214, 231, 310, 315–318,
 495, 619, 626, 648, 649–650, 656, 692, 694, 701, 714,
 720, 732, 734, 743, 966
 advantages to using, 650
 tryAdvance(), 526, 527–528, 987
 tryLock(), 917, 944
 trySplit(), 988–989
 tryUnfork(), 962
 Two's complement, 66–67
 TYPE, 443, 447, 455, 458, 460
 Type argument(s), 340, 342, 346
 and bounded types, 347–349
 and generic class hierarchies, 364
 and type inference, 358, 372–373
 Type conversion
 automatic, 35, 48, 130–131
 narrowing, 48
 widening, 48
 Type enumeration, 465
 Type interface, 496
 Type parameter(s)
 and bounded types, 346–349, 361–362
 cannot create an instance of a, 377
 and class hierarchies, 365–368
 and erasure, 341, 373
 and primitive types, 342
 and static members, 377
 and type safety, 342
 used with a class, 340, 345–346, 347
 used with a method, 340, 356–359
 Type safety
 and collections, 500, 550
 and generic methods, 359
 and generics, 337, 338, 341, 342–344, 500
 and raw types, 362–364
 and wildcard arguments, 349–352, 353
 type(), 465
 Type(s), 27
 annotations, 16, 292–297
 bounded, 347–349
 casting, 48–49, 50
 checking, 10, 11, 35, 341, 342–344, 363, 379
 class as a data, 109, 110, 113, 114, 116, 126
 inference, 358, 372–373, 383, 386, 395
 non-reifiable, 292
 parameterized, 338, 340
 promotion, 37, 50–51, 69–70
 raw, 362–364
 simple, 35
 TypeNotPresentException, 226
 Types, primitive (simple), 35–36, 114, 136, 272, 342
 autoboxing/unboxing and, 274–277, 279, 500,
 514
 and collections, 500, 514
 iterators for, 499
 to a string representation, converting, 417, 418,
 428–429
 to or from a sequence of bytes, converting, 667–
 669
 wrappers for, 272–274, 279, 342, 442–460
 Typesafe view of a collection, obtaining a dynamically,
 550
-
- U**
- UDP protocol, 727, 728, 742
 UI delegate, 1023, 1024
 ulp(), 478, 479
 UnaryOperator functional interface, 504, 639
 UnavailableException, 1217, 1220
 Unboxing, 274
 uncaughtException(), 495
 UncaughtExceptionHandler interface, 495
 Unchecked warnings and raw types, 364
 UnicastRemoteObject, 1006
 Unicode, 39, 40, 43, 302, 303, 415, 416, 420, 458, 670
 code points, table of some Character methods
 providing support for, 459
 support for 32-bit, 458
 Unicode Technical Standard (UTS) 35, 595
 Uniform Resource Identifier (URI), 741
 Uniform Resource Locator (URL). *See* URL (Uniform
 Resource Locator)
 UNIX, 4, 727
 UnknownHostException, 729, 730
 unlock(), 917, 944
 unmodifiable... collections methods, 554–555
 unordered(), 966, 977
 Unreachable code, 108, 219
 unread(), 662, 678
 UnsupportedOperationException, 226, 501, 502, 504,
 521, 531, 550, 699
 update(), 598, 599, 754, 756, 757, 811, 832
 overriding, 754
 URI (Uniform Resource Identifier), 741
 URI class, 741
 URL (Uniform Resource Locator), 735, 741, 760, 1211
 specification format, 735
 URL class, 735–736, 738, 739, 766
 methods, list of some, 737
 URLConnection class, 736–739, 741
 useDelimiter(), 628–629
 User Datagram Protocol (UDP), 727, 728, 742

useRadix(), 630
 UTS 35, 595
 UUID class, 635

V

value (annotation member name), 289, 290
 valueChanged(), 1058, 1059, 1063, 1064
 valueOf(), 266–267, 418, 428–429, 444, 446, 448, 449, 451, 453, 460, 492, 582
 values(), 266–267, 531, 533
 van Hoff, Arthur, 6
 Varargs, 14, 155–160
 and ambiguity, 159–160
 methods, overloading, 158–159
 and Java's printf(), 155
 parameter, 157, 521
 Variable(s), 44–48
 capture, 395–396
 declaration, 27, 29, 44–45, 46–48
 definition of, 26, 44
 dynamic initialization of, 45
 effectively final, 395–396
 enumeration, 264
 final, 147, 263
 instance. *See* Instance variables
 interface, 197, 204–206
 interface reference, 198–199, 204
 object reference. *See* Object reference variables
 scope and lifetime of, 45–48
 Vector class, 498–499, 511, 530, 561, 562–566
 legacy methods, table of, 563–564
 VetoableChangeListener interface, 1203, 1204
 Viewport, scroll pane, 1056, 1157
 visitFile(), 718, 719
 void, 25, 116
 Void class, 460
 volatile modifier, 322
 VSPACE, 760, 761

W

wait(), 185, 186, 251, 253–255, 258–259, 471, 915, 943, 944, 964
 waitFor(), 461, 464
 WAITING, 260
 WALL_TIME, 594
 walk(), 695
 walkFileTree(), 717–718
 Warth, Chris, 6
 Watchable interface, 694
 WeakHashMap class, 537
 Web browser, 767
 executing applet in, 320, 321, 747–748, 751, 753, 759, 760, 761, 766, 801
 using status window of, 759
 Web server and servlets, 1211, 1212
 WebView class, 1196
 weightx constraint field, 867, 868
 weighty constraint field, 867, 868
 while loop, 89–90

Whitespace, 32, 82
 from a string, removing, 428
 whois, 728, 732–734, 735
 WIDTH, 760, 761
 Wildcard arguments, 349–356, 370
 bounded, 352–356
 used in creating an array, 379
 Window, AWT-based
 displaying information within an, 811
 as an event source, 780–781, 782
 frame. *See* Frame window
 fundamentals, 800–801
 and graphics, 811
 status, using, 759–760
 Window class, 781, 800, 801, 876
 Window, Swing JFrame, 1028
 windowActivated(), 785
 WindowAdapter class, 792
 windowClosed(), 785
 windowClosing(), 785, 803
 WindowConstants interface, 1028
 windowDeactivated(), 785
 windowDeiconified(), 785
 WindowEvent class, 772, 774, 780–781
 WindowFocusListener interface, 782, 785, 792, 793
 windowGainedFocus(), 785
 windowIconified(), 785
 WindowListener interface, 782, 785, 792, 803
 windowLostFocus(), 785
 windowOpened(), 785
 WindowStateListener interface, 792
 Work stealing, 951, 962
 World Wide Web (WWW), 6, 7, 11, 735
 wrap(), 691
 Wrappers, primitive type, 272–274, 279, 342, 442–460
 WRITE, 465
 write(), 303, 304, 308, 314–315, 652, 672, 683, 684, 693, 706, 707, 711, 723, 724
 writeBoolean(), 668, 684
 writeDouble(), 668, 684
 Writer class, 303–304, 650, 670, 673, 688
 methods defined by, table of, 671–672
 writeExternal(), 683
 writeInt(), 668, 684
 writeObject(), 683, 686
 writeTo(), 659

X

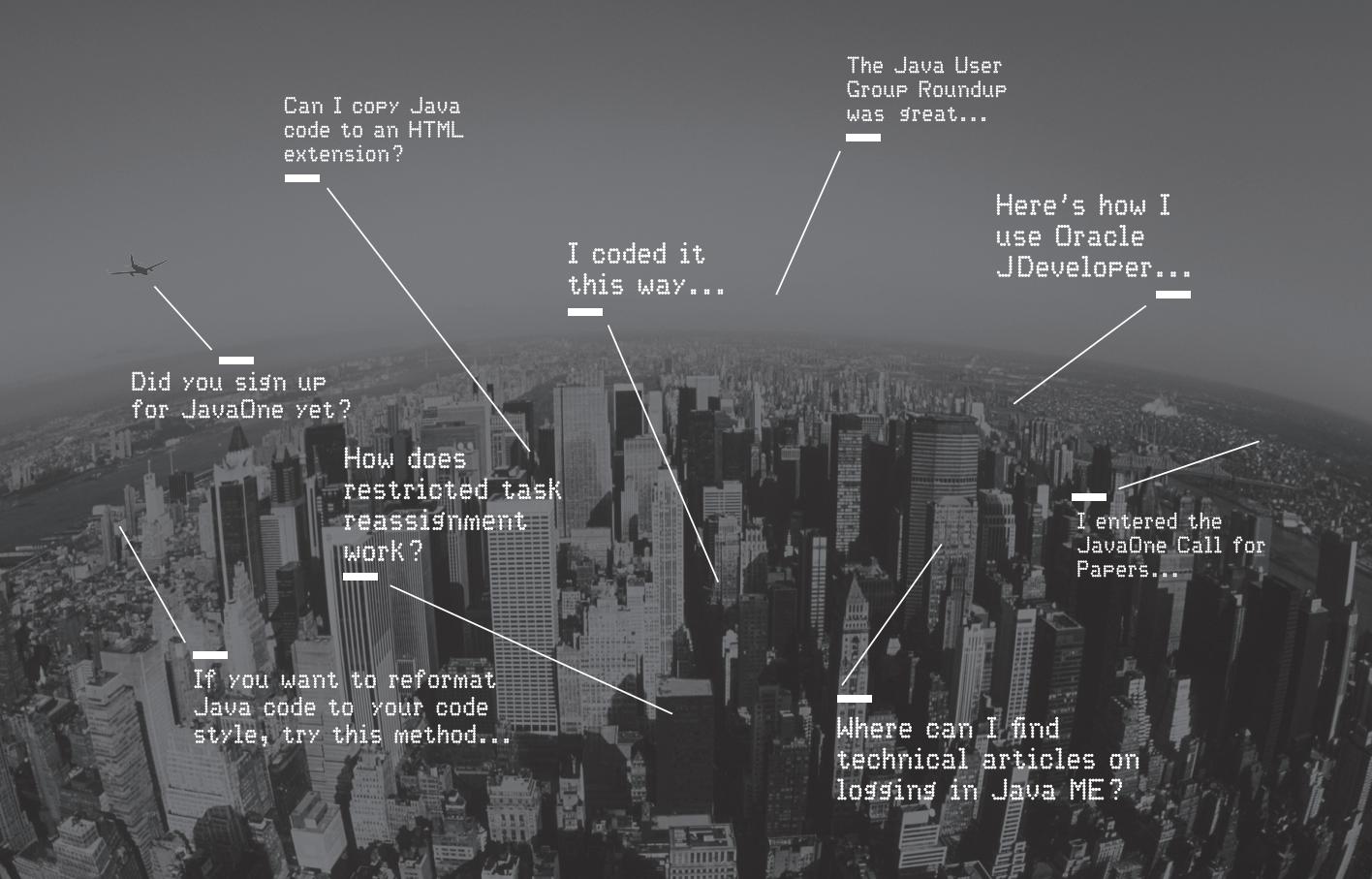
XOR (exclusive OR) operator (^)
 bitwise, 66, 67, 68–69
 Boolean logical, 75–76

Y

Yellin, Frank, 6

Z

Zero crossing, 67
 ZIP file format, 639



Oracle Technology Network. It's code for sharing expertise.

Come to the best place to collaborate with other IT professionals on everything Java.

Oracle Technology Network is the world's largest community of developers, administrators, and architects using Java and other industry-standard technologies with Oracle products.

Sign up for a free membership and you'll have access to:

- Discussion forums and hands-on labs
- Free downloadable software and sample code
- Product documentation
- Member-contributed content

Take advantage of our global network of knowledge.

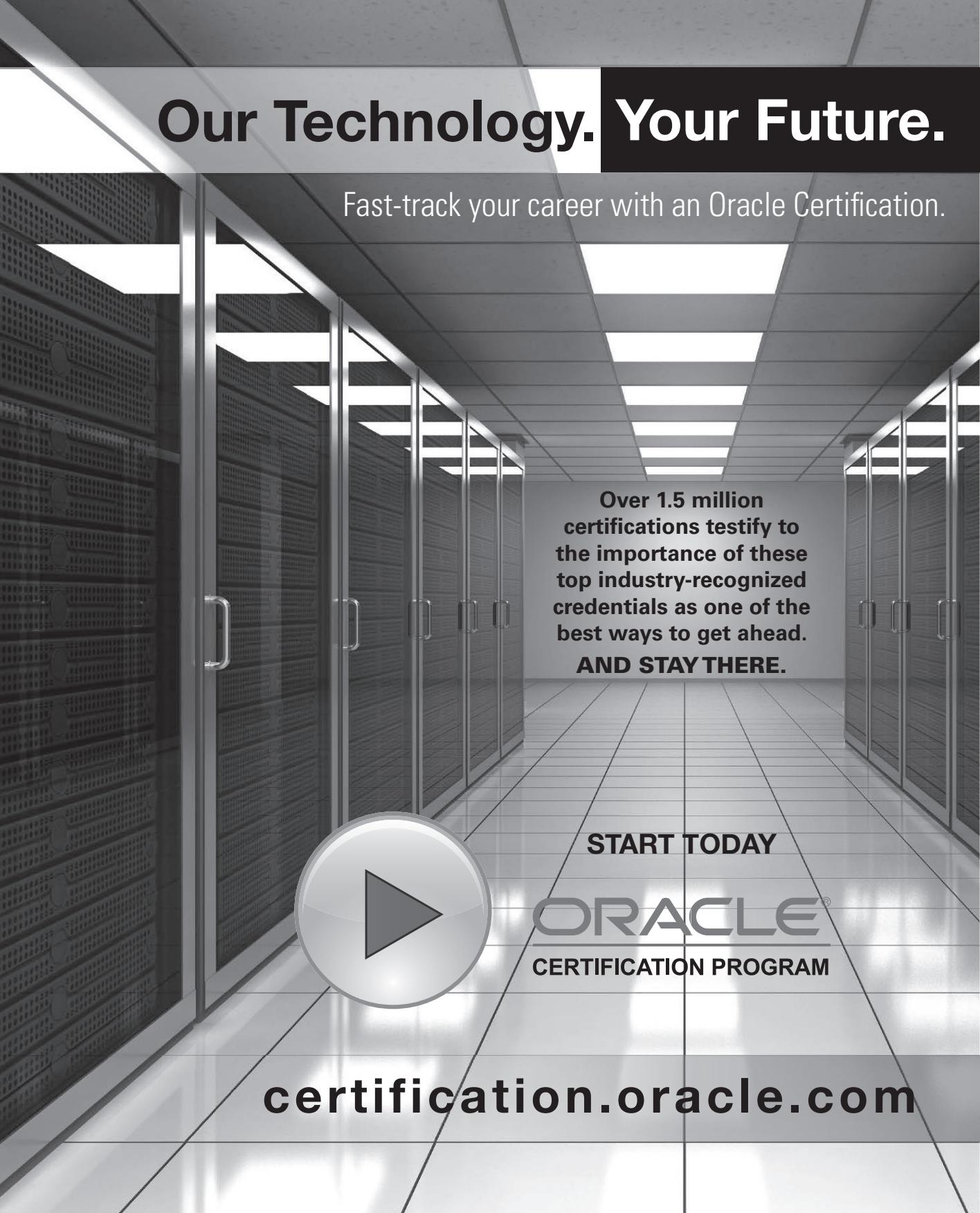
JOIN TODAY ▷ Go to: oracle.com/technetwork/java

ORACLE®
TECHNOLOGY NETWORK

ORACLE®

Our Technology. Your Future.

Fast-track your career with an Oracle Certification.



Over 1.5 million certifications testify to the importance of these top industry-recognized credentials as one of the best ways to get ahead.

AND STAY THERE.

START TODAY

ORACLE®
CERTIFICATION PROGRAM

certification.oracle.com



**Hardware and Software
Engineered to Work Together**



Stay Connected

oracle.com/technetwork/oracleace



Need help? Need consultation? Need an informed opinion?

You Need an Oracle ACE

Oracle partners, developers, and customers look to Oracle ACEs and Oracle ACE Directors for focused product expertise, systems and solutions discussion, and informed opinions on a wide range of data center implementations.

Their credentials are strong as Oracle product and technology experts, community enthusiasts, and solutions advocates.

And now is a great time to learn more about this elite group—or nominate a worthy colleague.

For more information about the Oracle ACE program, go to:
oracle.com/technetwork/oracleace





Reach More than 700,000 Oracle Customers with Oracle Publishing Group

Connect with the Audience
that Matters Most to Your Business



Oracle Magazine

The Largest IT Publication in the World

Circulation: 550,000

Audience: IT Managers, DBAs, Programmers, and Developers



Profit

Business Insight for Enterprise-Class Business Leaders to Help Them Build a Better Business Using Oracle Technology

Circulation: 100,000

Audience: Top Executives and Line of Business Managers



Java Magazine

The Essential Source on Java Technology, the Java Programming Language, and Java-Based Applications

Circulation: 125,000 and Growing Steady

Audience: Corporate and Independent Java Developers, Programmers, and Architects



For more information
or to sign up for a FREE
subscription:
Scan the QR code to visit
Oracle Publishing online.

ORACLE®