ECS-503 Object Oriented Techniques

UNIT-4 Part-2

CHAPTER

16 String Handling

Java implements strings as objects of type String. Implementing strings as built-in objects allows Java to provide a full complement of features that make string handling convenient. For example, Java has methods to compare two strings, search for a substring, concatenate two strings, and change the case of letters within a string. Also, String objects can be constructed a number of ways, making it easy to obtain a string when needed. Somewhat unexpectedly, when you create a String object, you are creating a string that cannot be changed. That is, once a String object has been created, you cannot change the characters that comprise that string. At first, this may seem to be a serious restriction.

However, such is not the case. You can still perform all types of string operations. The difference is that each time you need an altered version of an existing string, a new String object is created that contains the modifications. The original string is left unchanged. This approach is used because fixed, immutable strings can be implemented more efficiently than changeable ones. For those cases in which a modifiable string is desired, Java provides two options: StringBuffer and StringBuilder. Both hold strings that can be modified after they are created.

The String, StringBuffer, and StringBuilder classes are defined in java.lang. Thus, they are available to all programs automatically. All are declared final, which means that none of these classes may be subclassed. This allows certain optimizations that increase performance to take place on common string operations. All three implement the CharSequence interface.

One last point: To say that the strings within objects of type String are unchangeable means that the contents of the String instance cannot be changed after it has been created. However, a variable declared as a String reference can be changed to point at some other String object at any time.

The String Constructors

The String class supports several constructors. To create an empty String, call the default constructor. For example,

```java
String s = new String();
```

will create an instance of String with no characters in it.

Frequently, you will want to create strings that have initial values. The String class provides a variety of constructors to handle this. To create a String initialized by an array of characters, use the constructor shown here:

```java
String(char chars[])
```

Here is an example:

```java
char chars[] = { 'a', 'b', 'c' };
String s = new String(chars);
```

This constructor initializes s with the string "abc".

You can specify a subrange of a character array as an initializer using the following constructor:

```java
String(char chars[], int startIndex, int numChars)
```

Here, startIndex specifies the index at which the subrange begins, and numChars specifies the number of characters to use. Here is an example:

```java
char chars[] = { 'a', 'b', 'c', 'd', 'e', 'f' };
String s = new String(chars, 2, 3);
```
This initializes `s` with the characters `cde`.
You can construct a `String` object that contains the same character sequence as another `String` object using this constructor:

```java
String(String strObj)
```

Here, `strObj` is a `String` object. Consider this example:

```java
// Construct one String from another.
class MakeString {
    public static void main(String args[]) {
        char c[] = {'J', 'a', 'v', 'a'};
        String s1 = new String(c);
        String s2 = new String(s1);

        System.out.println(s1);
        System.out.println(s2);
    }
}
```

The output from this program is as follows:
As you can see, `s1` and `s2` contain the same string.

Even though Java’s `char` type uses 16 bits to represent the basic Unicode character set, the typical format for strings on the Internet uses arrays of 8-bit bytes constructed from the ASCII character set. Because 8-bit ASCII strings are common, the `String` class provides constructors that initialize a string when given a `byte` array. Two forms are shown here:

```java
String(byte chrs[])
String(byte chrs[], int startIndex, int numChars)
```

Here, `chrs` specifies the array of bytes. The second form allows you to specify a subrange. In each of these constructors, the byte-to-character conversion is done by using the default character encoding of the platform.

You can construct a `String` from a `StringBuffer` by using the constructor shown here:

```java
String(StringBuffer strBufObj)
```

You can construct a `String` from a `StringBuilder` by using this constructor:

```java
String(StringBuilder strBuildObj)
```

The following constructor supports the extended Unicode character set:

```java
String(int codePoints[], int startIndex, int numChars)
```

Here, `codePoints` is an array that contains Unicode code points. The resulting string is constructed from the range that begins at `startIndex` and runs for `numChars`.

There are also constructors that let you specify a `Charset`.

### String Length

The length of a string is the number of characters that it contains. To obtain this value, call the `length()` method, shown here:

```java
int length()
```

The following fragment prints "3", since there are three characters in the string `s`:

```java
char chars[] = {'a', 'b', 'c'};
String s = new String(chars);
System.out.println(s.length());
```
Special String Operations
Because strings are a common and important part of programming, Java has added special support for several string operations within the syntax of the language. These operations include the automatic creation of new String instances from string literals, concatenation of multiple String objects by use of the + operator, and the conversion of other data types to a string representation. There are explicit methods available to perform all of these functions, but Java does them automatically as a convenience for the programmer and to add clarity.

String Literals
The earlier examples showed how to explicitly create a String instance from an array of characters by using the new operator. However, there is an easier way to do this using a string literal. For each string literal in your program, Java automatically constructs a String object. Thus, you can use a string literal to initialize a String object. For example, the following code fragment creates two equivalent strings:

```java
char chars[] = { 'a', 'b', 'c' };
String s1 = new String(chars);

String s2 = "abc"; // use string literal
```

Because a String object is created for every string literal, you can use a string literal any place you can use a String object. For example, you can call methods directly on a quoted string as if it were an object reference, as the following statement shows. It calls the length() method on the string "abc". As expected, it prints "3".

```java
System.out.println("abc".length());
```

String Concatenation
In general, Java does not allow operators to be applied to String objects. The one exception to this rule is the + operator, which concatenates two strings, producing a String object as the result. This allows you to chain together a series of + operations. For example, the following fragment concatenates three strings:

```java
String age = "9";
String s = "He is " + age + " years old.";
System.out.println(s);
```

This displays the string "He is 9 years old."

One practical use of string concatenation is found when you are creating very long strings. Instead of letting long strings wrap around within your source code, you can break them into smaller pieces, using the + to concatenate them.

String Concatenation with Other Data Types
You can concatenate strings with other types of data. For example, consider this slightly different version of the earlier example:

```java
int age = 9;
String s = "He is " + age + " years old.";
System.out.println(s);
```

In this case, age is an int rather than another String, but the output produced is the same as before. This is because the int value in age is automatically converted into its string representation within a
String object. This string is then concatenated as before. The compiler will convert an operand to its string equivalent whenever the other operand of the + is an instance of String. Be careful when you mix other types of operations with string concatenation expressions, however. You might get surprising results. Consider the following:

```java
String s = "four: " + 2 + 2;
System.out.println(s);
```

This fragment displays

```
four: 22
```

rather than the

```
four: 4
```

that you probably expected. Here’s why. Operator precedence causes the concatenation of "four" with the string equivalent of 2 to take place first. This result is then concatenated with the string equivalent of 2 a second time. To complete the integer addition first, you must use parentheses, like this:

```java
String s = "four: " + (2 + 2);
```

Now s contains the string "four: 4".

**String Conversion and toString()**

When Java converts data into its string representation during concatenation, it does so by calling one of the overloaded versions of the string conversion method `valueOf()` defined by `String`. `valueOf()` is overloaded for all the primitive types and for type `Object`. For the primitive types, `valueOf()` returns a string that contains the human-readable equivalent of the value with which it is called. For objects, `valueOf()` calls the `toString()` method on the object. We will look more closely at `valueOf()` later in this chapter. Here, let’s examine the `toString()` method, because it is the means by which you can determine the string representation for objects of classes that you create. Every class implements `toString()` because it is defined by `Object`. However, the default implementation of `toString()` is seldom sufficient. For most important classes that you create, you will want to override `toString()` and provide your own string representations. Fortunately, this is easy to do. The `toString()` method has this general form:

```java
String toString()
```

To implement `toString()`, simply return a `String` object that contains the human-readable string that appropriately describes an object of your class. By overriding `toString()` for classes that you create, you allow them to be fully integrated into Java’s programming environment. For example, they can be used in `print()` and `println()` statements and in concatenation expressions. The following program demonstrates this by overriding `toString()` for the `Box` class:
// Override toString() for Box class.
class Box {
    double width;
    double height;
    double depth;

    Box(double w, double h, double d) {
        width = w;
        height = h;
        depth = d;
    }

    public String toString() {
        return "Dimensions are " + width + " by " + depth + " by " + height + ";";
    }
}

class toStringDemo {
    public static void main(String args[]) {
        Box b = new Box(10, 12, 14);
        String s = "Box b: " + b; // concatenate Box object
        System.out.println(b); // convert Box to string
        System.out.println(s);
    }
}

The output of this program is shown here:
Dimensions are 10.0 by 14.0 by 12.0
Box b: Dimensions are 10.0 by 14.0 by 12.0
As you can see, Box's toString() method is automatically invoked when a Box object is used in a concatenation expression or in a call to println().
As you can see, Box's toString() method is automatically invoked when a Box object is used in a concatenation expression or in a call to println().

Character Extraction
The String class provides a number of ways in which characters can be extracted from a String object. Several are examined here. Although the characters that comprise a string within a String object cannot be indexed as if they were a character array, many of the String methods employ an index (or offset) into the string for their operation. Like arrays, the string indexes begin at zero.

catAt()
To extract a single character from a String, you can refer directly to an individual character via the charAt() method. It has this general form:
char charAt(int where)
Here, where is the index of the character that you want to obtain. The value of where must be nonnegative and specify a location within the string. charAt() returns the character at the specified location. For example,

```java
    char ch;
    ch = "abc".charAt(1);
```
assigns the value b to ch.
getChars()
If you need to extract more than one character at a time, you can use the getChars() method. It has this general form:
void getChars(int sourceStart, int sourceEnd, char target[], int targetStart)
Here, sourceStart specifies the index of the beginning of the substring, and sourceEnd specifies an index that is one past the end of the desired substring. Thus, the substring contains the characters from sourceStart through sourceEnd–1. The array that will receive the characters is specified by target. The index within target at which the substring will be copied is passed in targetStart. Care must be taken to assure that the target array is large enough to hold the number of characters in the specified substring.

getBytes()
There is an alternative to getChars() that stores the characters in an array of bytes. This method is called getBytes(), and it uses the default character-to-byte conversions provided by the platform. Here is its simplest form:
byte[] getBytes()
Other forms of getBytes() are also available. getBytes() is most useful when you are exporting a String value into an environment that does not support 16-bit Unicode characters. For example, most Internet protocols and text file formats use 8-bit ASCII for all text interchange.

toCharArray()
If you want to convert all the characters in a String object into a character array, the easiest way is to call toCharArray(). It returns an array of characters for the entire string. It has this general form:
char[] toCharArray()
This function is provided as a convenience, since it is possible to use getChars() to achieve the same result.

String Comparison
The String class includes a number of methods that compare strings or substrings within strings. Several are examined here.
equals() and equalsIgnoreCase()
To compare two strings for equality, use equals(). It has this general form:
boolean equals(Object str)
Here, str is the String object being compared with the invoking String object. It returns true if the strings contain the same characters in the same order, and false otherwise. The comparison is case-sensitive. To perform a comparison that ignores case differences, call equalsIgnoreCase(). When it compares two strings, it considers A-Z to be the same as a-z. It has this general form:
boolean equalsIgnoreCase(String str)
Here, str is the String object being compared with the invoking String object. It, too, returns true if the strings contain the same characters in the same order, and false otherwise.

regionMatches()
The regionMatches() method compares a specific region inside a string with another specific region in another string. There is an overloaded form that allows you to ignore case in such comparisons. Here are the general forms for these two methods:
boolean regionMatches(int start1Index, String str2, int str2StartIndex, int numChars)
boolean regionMatches(boolean ignoreCase,
        int startIndex, String str2,
        int str2StartIndex, int numChars)

For both versions, startIndex specifies the index at which the region begins within the invoking String object. The String being compared is specified by str2. The index at which the comparison will start within str2 is specified by str2StartIndex. The length of the substring being compared is passed in numChars. In the second version, if ignoreCase is true, the case of the characters is ignored. Otherwise, case is significant.

startsWith( ) and endsWith( )
String defines two methods that are, more or less, specialized forms of regionMatches( ). The startsWith( ) method determines whether a given String begins with a specified string. Conversely, endsWith( ) determines whether the String in question ends with a specified string. They have the following general forms:
boolean startsWith(String str)
boolean endsWith(String str)

Here, str is the String being tested. If the string matches, true is returned. Otherwise, false is returned. For example,
"Foobar".endsWith("bar")
And
"Foobar".startsWith("Foo")
are both true.

A second form of startsWith( ), shown here, lets you specify a starting point:
boolean startsWith(String str, int startIndex)

Here, startIndex specifies the index into the invoking string at which point the search will begin. For example,
"Foobar".startsWith("bar", 3)
returns true.

equals( ) Versus ==
It is important to understand that the equals( ) method and the == operator perform two different operations. As just explained, the equals( ) method compares the characters inside a String object. The == operator compares two object references to see whether they refer to the same instance. The following program shows how two different String objects can contain the same characters, but references to these objects will not compare as equal:

compareTo( )
Often, it is not enough to simply know whether two strings are identical. For sorting applications, you need to know which is less than, equal to, or greater than the next. A string is less than another if it comes before the other in dictionary order. A string is greater than another if it comes after the other in dictionary order. The method compareTo( ) serves this purpose. It is specified by the Comparable<T> interface, which String implements. It has this general form:
int compareTo(String str)

Here, str is the String being compared with the invoking String. The result of the comparison is returned and is interpreted as shown here:

Value Meaning
Less than zero The invoking string is less than str.
Greater than zero The invoking string is greater than str.
Zero The two strings are equal.

**Searching Strings**
The `String` class provides two methods that allow you to search a string for a specified character or substring:

- `indexOf()` Searches for the first occurrence of a character or substring.
- `lastIndexOf()` Searches for the last occurrence of a character or substring.

These two methods are overloaded in several different ways. In all cases, the methods return the index at which the character or substring was found, or -1 on failure.

To search for the first occurrence of a character, use:

```java
int indexOf(int ch)
```

To search for the last occurrence of a character, use:

```java
int lastIndexOf(int ch)
```

Here, `ch` is the character being sought.

To search for the first or last occurrence of a substring, use:

```java
int indexOf(String str)
int lastIndexOf(String str)
```

Here, `str` specifies the substring.

You can specify a starting point for the search using these forms:

```java
int indexOf(int ch, int startIndex)
int lastIndexOf(int ch, int startIndex)
int indexOf(String str, int startIndex)
int lastIndexOf(String str, int startIndex)
```

Here, `startIndex` specifies the index at which point the search begins. For `indexOf()`, the search runs from `startIndex` to the end of the string. For `lastIndexOf()`, the search runs from `startIndex` to zero.

**Modifying a String**
Because `String` objects are immutable, whenever you want to modify a `String`, you must either copy it into a `StringBuffer` or `StringBuilder`, or use a `String` method that constructs a new copy of the string with your modifications complete. A sampling of these methods are described here.

**substring()**
You can extract a substring using `substring()`. It has two forms. The first is:

```java
String substring(int startIndex)
```

Here, `startIndex` specifies the index at which the substring will begin. This form returns a copy of the substring that begins at `startIndex` and runs to the end of the invoking string.

The second form of `substring()` allows you to specify both the beginning and ending index of the substring:

```java
String substring(int startIndex, int endIndex)
```

Here, `startIndex` specifies the beginning index, and `endIndex` specifies the stopping point. The string returned contains all the characters from the beginning index, up to, but not including, the ending index.

**concat()**
You can concatenate two strings using `concat()`, shown here:

```java
String concat(String str)
```

This method creates a new object that contains the invoking string with the contents of `str` appended to the end. `concat()` performs the same function as `+`

For example,
String s1 = "one";
String s2 = s1.concat("two");
puts the string "onetwo" into s2. It generates the same result as the following sequence:
  String s1 = "one";
  String s2 = s1 + "two";

replace()
The replace( ) method has two forms. The first replaces all occurrences of one character in the
invoking string with another character. It has the following general form:
String replace(char original, char replacement)
Here, original specifies the character to be replaced by the character specified by replacement. The
resulting string is returned. For example,
  String s = "Hello".replace('l', 'w');
puts the string "Hewwo" into s.
The second form of replace( ) replaces one character sequence with another. It has this general
form:
String replace( CharSequence original, CharSequence replacement)

trim()
The trim( ) method returns a copy of the invoking string from which any leading and trailing
whitespace has been removed. It has this general form:
String trim( )
Here is an example:
  String s = " Hello World " .trim();
This puts the string "Hello World" into s.
The trim( ) method is quite useful when you process user commands. For example, the following
program prompts the user for the name of a state and then displays that state's capital. It uses trim( )
to remove any leading or trailing whitespace that may have inadvertently been entered by the
user.

Data Conversion Using valueOf( )
The valueOf( ) method converts data from its internal format into a human-readable form. It is a
static method that is overloaded within String for all of Java's built-in types so that each type can be
converted properly into a string. valueOf( ) is also overloaded for type Object, so an object of any
class type you create can also be used as an argument. (Recall that Object is a superclass for all
classes.) Here are a few of its forms:
static String valueOf(double num)
static String valueOf(long num)
static String valueOf(Object ob)
static String valueOf(char chars[ ])
As discussed earlier, valueOf( ) is called when a string representation of some other type of data is
needed—for example, during concatenation operations. You can call this method directly with any
data type and get a reasonable String representation. All of the simple types are converted to their
common String representation. Any object that you pass to valueOf( ) will return the result of a call
to the object's toString( ) method. In fact, you could just call toString( ) directly and get the same
result. For most arrays, valueOf( ) returns a rather cryptic string, which indicates that it is an array of
some type. For arrays of char, however, a String object is created that contains the characters in the
char array. There is a special version of valueOf( ) that allows you to specify a subset of a char array.
It has this general form:
static String valueOf(char chars[ ], int startIndex, int numChars)
Here, `chars` is the array that holds the characters, `startIndex` is the index into the array of characters at which the desired substring begins, and `numChars` specifies the length of the substring.

### Changing the Case of Characters Within a String

The method `toLowerCase()` converts all the characters in a string from uppercase to lowercase. The `toUpperCase()` method converts all the characters in a string from lowercase to uppercase. Nonalphabetical characters, such as digits, are unaffected. Here are the simplest forms of these methods:

```java
String toLowerCase()
String toUpperCase()
```

Both methods return a `String` object that contains the uppercase or lowercase equivalent of the invoking `String`. The default locale governs the conversion in both cases.

### StringBuffer

`StringBuffer` supports a modifiable string. As you know, `String` represents fixed-length, immutable character sequences. In contrast, `StringBuffer` represents growable and writable character sequences. `StringBuffer` may have characters and substrings inserted in the middle or appended to the end. `StringBuffer` will automatically grow to make room for such additions and often has more characters preallocated than are actually needed, to allow room for growth.

#### StringBuffer Constructors

`StringBuffer` defines these four constructors:

```java
StringBuffer()
StringBuffer(int size)
StringBuffer(String str)
StringBuffer(CharSequence chars)
```

The default constructor (the one with no parameters) reserves room for 16 characters without reallocation. The second version accepts an integer argument that explicitly sets the size of the buffer. The third version accepts a `String` argument that sets the initial contents of the `StringBuffer` object and reserves room for 16 more characters without reallocation. `StringBuffer` allocates room for 16 additional characters when no specific buffer length is requested, because reallocation is a costly process in terms of time. Also, frequent reallocations can fragment memory. By allocating room for a few extra characters, `StringBuffer` reduces the number of reallocations that take place. The fourth constructor creates an object that contains the character sequence contained in `chars` and reserves room for 16 more characters.

#### length() and capacity()

The current length of a `StringBuffer` can be found via the `length()` method, while the total allocated capacity can be found through the `capacity()` method. They have the following general forms:

```java
int length()
int capacity()
```

Here is an example:

```java
// StringBuffer length vs. capacity.
class StringBufferDemo {
   public static void main(String args[]) {
      StringBuffer sb = new StringBuffer("Hello");
      System.out.println("buffer = " + sb);
      System.out.println("length = " + sb.length());
      System.out.println("capacity = " + sb.capacity());
   }
}
```
Here is the output of this program, which shows how StringBuffer reserves extra space for additional manipulations:

```java
Buffer = Hello
length = 5
capacity = 21
```

Since `sb` is initialized with the string "Hello" when it is created, its length is 5. Its capacity is 21 because room for 16 additional characters is automatically added.

**ensureCapacity()**

If you want to preallocate room for a certain number of characters after a StringBuffer has been constructed, you can use `ensureCapacity()` to set the size of the buffer. This is useful if you know in advance that you will be appending a large number of small strings to a StringBuffer.

`ensureCapacity()` has this general form:

```java
void ensureCapacity(int minCapacity)
```

Here, `minCapacity` specifies the minimum size of the buffer. (A buffer larger than `minCapacity` may be allocated for reasons of efficiency.)

**setLength()**

To set the length of the string within a StringBuffer object, use `setLength()`.

```java
void setLength(int len)
```

Here, `len` specifies the length of the string. This value must be nonnegative. When you increase the size of the string, null characters are added to the end. If you call `setLength()` with a value less than the current value returned by `length()`, then the characters stored beyond the new length will be lost. The `setCharAtDemo` sample program in the following section uses `setLength()` to shorten a StringBuffer.

**charAt() and setCharAt()**

The value of a single character can be obtained from a StringBuffer via the `charAt()` method. You can set the value of a character within a StringBuffer using `setCharAt()`.

```java
char charAt(int where)
void setCharAt(int where, char ch)
```

For `charAt()`, `where` specifies the index of the character being obtained. For `setCharAt()`, `where` specifies the index of the character being set, and `ch` specifies the new value of that character. For both methods, `where` must be nonnegative and must not specify a location beyond the end of the string.

**getChars()**

To copy a substring of a StringBuffer into an array, use the `getChars()` method.

```java
void getChars(int sourceStart, int sourceEnd, char target[], int targetStart)
```

Here, `sourceStart` specifies the index of the beginning of the substring, and `sourceEnd` specifies an index that is one past the end of the desired substring. This means that the substring contains the characters from `sourceStart` through `sourceEnd−1`. The array that will receive the characters is specified by `target`. The index within `target` at which the substring will be copied is passed in `targetStart`.

Care must be taken to assure that the `target` array is large enough to hold the number of characters in the specified substring.
append( )
The `append( )` method concatenates the string representation of any other type of data to the end of the invoking `StringBuffer` object. It has several overloaded versions. Here are a few of its forms:
- `StringBuffer append(String str)`
- `StringBuffer append(int num)`
- `StringBuffer append(Object obj)`
The string representation of each parameter is obtained, often by calling `String.valueOf( )`. The result is appended to the current `StringBuffer` object. The buffer itself is returned by each version of `append( )`. This allows subsequent calls to be chained together.

insert( )
The `insert( )` method inserts one string into another. It is overloaded to accept values of all the primitive types, plus `Strings`, `Objects`, and `CharSequences`. Like `append( )`, it obtains the string representation of the value it is called with. This string is then inserted into the invoking `StringBuffer` object. These are a few of its forms:
- `StringBuffer insert(int index, String str)`
- `StringBuffer insert(int index, char ch)`
- `StringBuffer insert(int index, Object obj)`
Here, `index` specifies the index at which point the string will be inserted into the invoking `StringBuffer` object.

reverse( )
You can reverse the characters within a `StringBuffer` object using `reverse( )`, shown here:
- `StringBuffer reverse( )`

delete( ) and deleteCharAt( )
You can delete characters within a `StringBuffer` by using the methods `delete( )` and `deleteCharAt( )`. These methods are shown here:
- `StringBuffer delete(int startIndex, int endIndex)`
- `StringBuffer deleteCharAt(int loc)`
The `delete( )` method deletes a sequence of characters from the invoking object. Here, `startIndex` specifies the index of the first character to remove, and `endIndex` specifies an index past the last character to remove. Thus, the substring deleted runs from `startIndex` to `endIndex`–1. The resulting `StringBuffer` object is returned. The `deleteCharAt( )` method deletes the character at the index specified by `loc`. It returns the resulting `StringBuffer` object.

replace( )
You can replace one set of characters with another set inside a `StringBuffer` object by calling `replace( )`. Its signature is shown here:
- `StringBuffer replace(int startIndex, int endIndex, String str)`
The substring being replaced is specified by the indexes `startIndex` and `endIndex`. Thus, the substring at `startIndex` through `endIndex`–1 is replaced. The replacement string is passed in `str`. The resulting `StringBuffer` object is returned.

substring( )
You can obtain a portion of a `StringBuffer` by calling `substring( )`. It has the following two forms:
- `String substring(int startIndex)`
- `String substring(int startIndex, int endIndex)`
The first form returns the substring that starts at `startIndex` and runs to the end of the invoking `StringBuilder` object. The second form returns the substring that starts at `startIndex` and runs through `endIndex` – 1.

CHAPTER

22 Networking

Networking Basics
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:

<table>
<thead>
<tr>
<th>Authenticator</th>
<th>InetAddress</th>
<th>SocketAddress</th>
</tr>
</thead>
<tbody>
<tr>
<td>CacheRequest</td>
<td>IntetSocketAddress</td>
<td>SocketImpl</td>
</tr>
<tr>
<td>CacheResponse</td>
<td>InterfaceAddress</td>
<td>SocketPermission</td>
</tr>
<tr>
<td>ContentHandler</td>
<td>JarURLConnection</td>
<td>StandardSocketOption</td>
</tr>
<tr>
<td>CookieHandler</td>
<td>MulticastSocket</td>
<td>URI</td>
</tr>
<tr>
<td>CookieManager</td>
<td>NetPermission</td>
<td>URL</td>
</tr>
<tr>
<td>DatagramPacket</td>
<td>NetworkInterface</td>
<td>URLClassLoader</td>
</tr>
<tr>
<td>DatagramSocket</td>
<td>PasswordAuthentication</td>
<td>URLConnection</td>
</tr>
<tr>
<td>DatagramSocketImpl</td>
<td>Proxy</td>
<td>URLEncoder</td>
</tr>
<tr>
<td>HttpClient</td>
<td>ProxySelector</td>
<td>URLDecoder</td>
</tr>
<tr>
<td>HttpURLConnection</td>
<td>ResponseCache</td>
<td>URLPermission (Added by JDK 8.)</td>
</tr>
<tr>
<td>IDN</td>
<td>SecureCacheResponse</td>
<td>URLStreamHandler</td>
</tr>
<tr>
<td>InetAddress</td>
<td>ServerSocket</td>
<td>Socket</td>
</tr>
<tr>
<td>InetAddress[]</td>
<td>getAllByName(String hostName)</td>
<td>throws UnknownHostException</td>
</tr>
</tbody>
</table>

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:

```java
// 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:

- **Boolean equals(Object other)** Returns true if this object has the same Internet address as other.
- **byte[] getAddress( )** Returns a byte array that represents the object’s IP address in network byte order.
- **String getHostAddress( )** Returns a string that represents the host address associated with the InetAddress object.
- **String getHostName( )** Returns a string that represents the host name associated with the InetAddress object.
- **Boolean isMulticastAddress( )** Returns true if this address is a multicast address. Otherwise, it returns false.
- **String toString( )** Returns a string that lists the host name and the IP address for convenience.

Internet addresses are looked up in a series of hierarchically cached servers. That means that your local computer might know a particular name-to-IP-address mapping automatically, such as for itself and nearby servers. For other names, it may ask a local DNS server for IP address information. If that server doesn’t have a particular address, it can go to a remote site and ask for it. This can continue
all the way up to the root server. This process might take a long time, so it is wise to structure your code so that you cache IP address information locally rather than look it up repeatedly.

**Inet4Address and Inet6Address**
Java includes support for both IPv4 and IPv6 addresses. Because of this, two subclasses of `InetAddress` were created: `Inet4Address` and `Inet6Address`. `Inet4Address` represents a traditional-style IPv4 address. `Inet6Address` encapsulates a newer IPv6 address. Because they are subclasses of `InetAddress`, an `InetAddress` reference can refer to either. This is one way that Java was able to add IPv6 functionality without breaking existing code or adding many more classes. For the most part, you can simply use `InetAddress` when working with IP addresses because it can accommodate both styles.

**TCP/IP Client Sockets**
TCP/IP sockets are used to implement reliable, bidirectional, persistent, point-to-point, stream-based connections between hosts on the Internet. A socket can be used to connect Java’s I/O system to other programs that may reside either on the local machine or on any other machine on the Internet.

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:

```java
Socket(String hostName, int port) throws UnknownHostException, IOException
Creates a socket connected to the named host and port.

Socket(InetAddress ipAddress, int port) throws IOException
Creates a socket using a pre-existing InetAddress object and a port.
```

A `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:

```java
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.

```java
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.

```java
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()`.

**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/](http://www.MHProfessional.com/) and [http://www.MHProfessional.com:80/index.htm](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/](http://www.MHProfessional.com/) is the same as [http://www.MHProfessional.com/index.htm](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:

```java
URL(String urlSpecifier) throws MalformedURLException
```

The next two forms of the constructor allow you to break up the URL into its component parts:

```java
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.

```java
URL(URL urlObj, String urlSpecifier) throws MalformedURLException
```

**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:

```java
int getContentLength( ) Returns the size in bytes of the content associated with the resource. If the length is unavailable, −1 is returned.
Long getContentLengthLong( ) Returns the size in bytes of the content associated with the resource. If the length is unavailable, −1 is returned.
String getContentType( ) Returns the type of content found in the resource. This is the value of the content-type header field. Returns null if the content type is not available.
long getDate( ) Returns the time and date of the response represented in terms of milliseconds since January 1, 1970 GMT.
long getExpiration( ) Returns the expiration time and date of the resource represented in terms of milliseconds since January 1, 1970 GMT. Zero is returned if the expiration date is unavailable.
String getHeaderField(int idx) Returns the value of the header field at index idx. (Header field indexes begin at 0.) Returns null if the value of idx exceeds the number of fields.
String getHeaderField(String fieldName) Returns the value of header field whose name is specified by fieldName. Returns null if the specified name is not found.
String getHeaderFieldKey(int idx) Returns the header field key at index idx. (Header field indexes begin at 0.) Returns null if the value of idx exceeds the number of fields.
Map<String, List<String>> getHeaderFields( ) Returns a map that contains all of the header fields and values.
long getLastModified( ) Returns the time and date, represented in terms of milliseconds since January 1, 1970 GMT, of the last modification of the resource. Zero is returned if the lastmodified date is unavailable.
InputStream getInputStream( ) throws IOException Returns an InputStream that is linked to the resource. This stream can be used to obtain the content of the resource.

Notice that **URLConnection** defines several methods that handle header information. A header consists of pairs of keys and values represented as strings. By using **getHeaderField( )**, you can obtain the value associated with a header key. By calling **getHeaderFields( )**, you can obtain a map that contains all of the headers. Several standard header fields are available directly through methods such as **getDate( )** and **getContentType( )**.

**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:

```java
static Boolean getFollowRedirects( ) Returns true if redirects are automatically followed and false otherwise. This feature is on by default.
String getRequestMethod( ) Returns a string representing how URL requests are made. The default is GET. Other options, such as POST, are available.
int getResponseCode( ) throws IOException Returns the HTTP response code. −1 is returned if no response code can be obtained. An IOException is thrown if the connection fails.
String getResponseMessage( ) throws IOException Returns the response message associated with the response code. Returns null if no message is available. An IOException is thrown if the connection fails.
```
static void setFollowRedirects(Boolean how) If how is true, then redirects are automatically followed. If how is false, redirects are not automatically followed. By default, redirects are automatically followed. 

Void setRequestMethod(String how) throws ProtocolException Sets the method by which HTTP requests are made to that specified by how. The default method is GET, but other options, such as POST, are available. If how is invalid, a ProtocolException is thrown.

The 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. 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:

ServerSocket(int port) throws IOException Creates server socket on the specified port with a queue length of 50.
ServerSocket(int port, int maxQueue) throws IOException Creates a server socket on the specified port with a maximum queue length of maxQueue.
ServerSocket(int port, int maxQueue, InetAddress localAddress) throws IOException Creates a server socket on the specified port with a maximum queue length of maxQueue. On a multihomed host, localAddress specifies the IP address to which this socket binds.

ServerSocket has a method called accept(), which is a blocking call that will wait for a client to initiate communications and then return with a normal Socket that is then used for communication with the client.

Datagrams
TCP/IP-style networking is appropriate for most networking needs. It provides a serialized, predictable, reliable stream of packet data. This is not without its cost, however. TCP includes many complicated algorithms for dealing with congestion control on crowded networks, as well as pessimistic expectations about packet loss. This leads to a somewhat inefficient way to transport data. Datagrams provide an alternative. Datagrams are bundles of information passed between machines. They are somewhat like a hard throw from a well-trained but blindfolded catcher to the third baseman. Once the datagram has been released to its intended target, there is no assurance that it will arrive or even that someone will be there to catch it. Likewise, when the datagram is received, there is no assurance that it hasn’t been damaged in transit or that whoever sent it is still there to receive a response.
Java implements datagrams on top of the UDP protocol by using two classes: the `DatagramPacket` object is the data container, while the `DatagramSocket` is the mechanism used to send or receive the `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(InetSocketAddress 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 setTimeout(int millis)` throws `SocketException`

Sets the time-out period to the number of milliseconds passed in `millis`.

**DatagramPacket**

`DatagramPacket` defines several constructors. Four are shown here:

- `DatagramPacket(byte data [ ], int size)`
- `DatagramPacket(byte data [ ], int offset, int size)`
- `DatagramPacket(byte data [ ], int size, InetAddress ipAddress, int port)`
- `DatagramPacket(byte data [ ], int offset, int size, InetAddress ipAddress, int port)`

The first constructor specifies a buffer that will receive data and the size of a packet. It is used for receiving data over a `DatagramSocket`. The second form allows you to specify an offset into the buffer at which data will be stored. The third form specifies a target address and port, which are used by a `DatagramSocket` to determine where the data in the packet will be sent. The fourth form
transmits packets beginning at the specified offset into the data. Think of the first two forms as building an “in box,” and the second two forms as stuffing and addressing an envelope. 

**DatagramPacket** defines several methods, including those shown here, that give access to the address and port number of a packet, as well as the raw data and its length.

- ` InetAddress getAddress( )` Returns the address of the source (for datagrams being received) or destination (for datagrams being sent).
- ` byte[ ] getData( )` Returns the byte array of data contained in the datagram. Mostly used to retrieve data from the datagram after it has been received.
- ` int getLength( )` Returns the length of the valid data contained in the byte array that would be returned from the `getData( )` method. This may not equal the length of the whole byte array.
- ` int getOffset( )` Returns the starting index of the data.
- ` int getPort( )` Returns the port number.

### Void setAddress(InetAddress ipAddress)`
Sets the address to which a packet will be sent. The address is specified by `ipAddress`.

### void setData(byte[ ] data)`
Sets the data to `data`, the offset to zero, and the length to number of bytes in `data`.

### void setData(byte[ ] data, int idx, int size)`
Sets the data to `data`, the offset to `idx`, and the length to `size`.

### void setLength(int size)`
Sets the length of the packet to `size`.

### void setPort(int port)`
Sets the port to `port`.

---

**CHAPTER 23 The Applet Class**

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.

### Applet Basics

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 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.)

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>void destroy()</td>
<td>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.</td>
</tr>
<tr>
<td>AccessibleContext getAccessibleContext()</td>
<td>Returns the accessibility context for the invoking object.</td>
</tr>
<tr>
<td>AppletContext getAppletContext()</td>
<td>Returns the context associated with the applet.</td>
</tr>
<tr>
<td>String getAppletInfo()</td>
<td>Overrides this method should return a string that describes the applet. The default implementation returns null.</td>
</tr>
<tr>
<td>AudioClip getAudioClip(URL url)</td>
<td>Returns an AudioClip object that encapsulates the audio clip found at the location specified by url.</td>
</tr>
<tr>
<td>AudioClip getAudioClip(URL url, String clipName)</td>
<td>Returns an AudioClip object that encapsulates the audio clip found at the location specified by url and having the name specified by clipName.</td>
</tr>
<tr>
<td>URL getCodeBase()</td>
<td>Returns the URL associated with the invoking applet.</td>
</tr>
<tr>
<td>URL getDocumentBase()</td>
<td>Returns the URL of the HTML document that invokes the applet.</td>
</tr>
<tr>
<td>Image getImage(URL url)</td>
<td>Returns an Image object that encapsulates the image found at the location specified by url.</td>
</tr>
<tr>
<td>Image getImage(URL url, String imageNamed)</td>
<td>Returns an Image object that encapsulates the image found at the location specified by url and having the name specified by imageNamed.</td>
</tr>
<tr>
<td>Locale getLocale()</td>
<td>Returns a Locale object that is used by various locale-sensitive classes and methods.</td>
</tr>
<tr>
<td>String getParameter(String parameterName)</td>
<td>Returns the parameter associated with parameterName. null is returned if the specified parameter is not found.</td>
</tr>
</tbody>
</table>
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 runtime 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. 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:

```java
void init()
void start()
void stop()
void destroy()
void paint(Graphics g)
```
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. 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:

```java
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.

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:

```java
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.gray
- Color.red
- Color.green
- Color.pink
- Color.darkGray
- Color.magenta
- Color.magenta
- Color.green
- Color.green
- Color.pink
- Color.magenta

Uppercase versions of the constants are also defined. The following example sets the background color to green and the text color to red:

```java
setBackground(Color.green);
setForeground(Color.red);
```

**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:

```java
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 y, 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.

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:

```java
/* 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:

![Applet Viewer: SimpleBanner](image)
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 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 = applletFile
    [ALT = alternateText]
    [NAME = applletInstanceName]
    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>
```
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.

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.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Applet getApplet(String appletName)</td>
<td>Returns the applet specified by appletName if it is within the current applet context. Otherwise, null is returned.</td>
</tr>
<tr>
<td>Enumeration&lt;Applet&gt; getApplets()</td>
<td>Returns an enumeration that contains all of the applets within the current applet context.</td>
</tr>
<tr>
<td>AudioClip getAudioClip(URL url)</td>
<td>Returns an AudioClip object that encapsulates the audio clip found at the location specified by url.</td>
</tr>
<tr>
<td>Image getImage(URL url)</td>
<td>Returns an Image object that encapsulates the image found at the location specified by url.</td>
</tr>
<tr>
<td>InputStream getStream(String key)</td>
<td>Returns the stream linked to key. Keys are linked to streams by using the setStream() method. A null reference is returned if no stream is linked to key.</td>
</tr>
<tr>
<td>Iterator&lt;String&gt; getStreamKeys()</td>
<td>Returns an iterator for the keys associated with the invoking object. The keys are linked to streams. See getStream() and setStream().</td>
</tr>
<tr>
<td>void setStream(String key, InputStream stream) throws IOException</td>
<td>Links the stream specified by stream to the key passed in key. The key is deleted from the invoking object if stream is null.</td>
</tr>
<tr>
<td>void showDocument(URL url)</td>
<td>Brings the document at the URL specified by url into view. This method may not be supported by applet viewers.</td>
</tr>
</tbody>
</table>

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.

CHAPTER 24 Event Handling

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 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. The examples provided in the remainder of this book make frequent use of these techniques.

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.

<table>
<thead>
<tr>
<th>Event Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ActionEvent</td>
<td>Generated when a button is pressed, a list item is double-clicked, or a menu item is selected.</td>
</tr>
<tr>
<td>AdjustmentEvent</td>
<td>Generated when a scroll bar is manipulated.</td>
</tr>
<tr>
<td>ComponentEvent</td>
<td>Generated when a component is hidden, moved, resized, or becomes visible.</td>
</tr>
<tr>
<td>ContainerEvent</td>
<td>Generated when a component is added to or removed from a container.</td>
</tr>
<tr>
<td>FocusEvent</td>
<td>Generated when a component gains or loses keyboard focus.</td>
</tr>
<tr>
<td>InputEvent</td>
<td>Abstract superclass for all component input event classes.</td>
</tr>
<tr>
<td>ItemEvent</td>
<td>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.</td>
</tr>
<tr>
<td>KeyEvent</td>
<td>Generated when input is received from the keyboard.</td>
</tr>
<tr>
<td>MouseEvent</td>
<td>Generated when the mouse is dragged, moved, clicked, pressed, or released; also generated when the mouse enters or exits a component.</td>
</tr>
<tr>
<td>MouseWheelEvent</td>
<td>Generated when the mouse wheel is moved.</td>
</tr>
<tr>
<td>TextEvent</td>
<td>Generated when the value of a text area or text field is changed.</td>
</tr>
<tr>
<td>WindowEvent</td>
<td>Generated when a window is activated, closed, deactivated, deiconified, iconified, opened, or quit.</td>
</tr>
</tbody>
</table>
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:

```java
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. It is shown here:

```java
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 int 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:

```java
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:

```java
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:

```java
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:

```java
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:

<table>
<thead>
<tr>
<th>Modifier</th>
<th>Constant</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALT_MASK</td>
<td>BUTTON2_MASK</td>
</tr>
<tr>
<td>ALT_GRAPH_MASK</td>
<td>BUTTON3_MASK</td>
</tr>
<tr>
<td>BUTTON1_MASK</td>
<td>CTRL_MASK</td>
</tr>
<tr>
<td>META_MASK</td>
<td>SHIFT_MASK</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>Modifier</th>
<th>Constant</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALT_DOWN_MASK</td>
<td>BUTTON2_DOWN_MASK</td>
</tr>
<tr>
<td>ALT_GRAPH_DOWN_MASK</td>
<td>BUTTON3_DOWN_MASK</td>
</tr>
<tr>
<td>BUTTON1_DOWN_MASK</td>
<td>CTRL_DOWN_MASK</td>
</tr>
<tr>
<td>META_DOWN_MASK</td>
<td>SHIFT_DOWN_MASK</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>VK constants</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>VK_ALT</td>
<td>Virtual key code for the Alt key</td>
</tr>
<tr>
<td>VK_CANCEL</td>
<td>Virtual key code for the Cancel key</td>
</tr>
<tr>
<td>VK_CONTROL</td>
<td>Virtual key code for the Control key</td>
</tr>
<tr>
<td>VK_DOWN</td>
<td>Virtual key code for the Down key</td>
</tr>
<tr>
<td>VK_ESCAPE</td>
<td>Virtual key code for the Escape key</td>
</tr>
<tr>
<td>VK_LEFT</td>
<td>Virtual key code for the Left key</td>
</tr>
<tr>
<td>VK_PAGE_DOWN</td>
<td>Virtual key code for the Page Down key</td>
</tr>
<tr>
<td>VK_PAGE_UP</td>
<td>Virtual key code for the Page Up key</td>
</tr>
<tr>
<td>VK_PRINT</td>
<td>Virtual key code for the Print key</td>
</tr>
<tr>
<td>VK_RIGHT</td>
<td>Virtual key code for the Right key</td>
</tr>
<tr>
<td>VK_SHIFT</td>
<td>Virtual key code for the Shift key</td>
</tr>
<tr>
<td>VK.Typed</td>
<td>Virtual key code for the Typing key</td>
</tr>
<tr>
<td>VK.UP</td>
<td>Virtual key code for the Up key</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>NOBUTTON</th>
<th>BUTTON1</th>
<th>BUTTON2</th>
<th>BUTTON3</th>
</tr>
</thead>
</table>

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:
**WindowEvent** is a subclass of **ComponentEvent**. It defines several constructors. The first is

```java
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:

```java
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:

```java
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:

```java
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.

<table>
<thead>
<tr>
<th>Event Source</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Button</td>
<td>Generates action events when the button is pressed.</td>
</tr>
<tr>
<td>Check box</td>
<td>Generates item events when the check box is selected or deselected.</td>
</tr>
<tr>
<td>Choice</td>
<td>Generates item events when the choice is changed.</td>
</tr>
<tr>
<td>List</td>
<td>Generates action events when an item is double-clicked; generates item events when an item is selected or deselected.</td>
</tr>
<tr>
<td>Menu item</td>
<td>Generates action events when a menu item is selected; generates item events when a checkable menu item is selected or deselected.</td>
</tr>
<tr>
<td>Scroll bar</td>
<td>Generates adjustment events when the scroll bar is manipulated.</td>
</tr>
<tr>
<td>Text component</td>
<td>Generates text events when the user enters a character.</td>
</tr>
<tr>
<td>Window</td>
<td>Generates window events when a window is activated, closed, deactivated, deiconified, iconified, opened, or quit.</td>
</tr>
</tbody>
</table>
**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 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.

<table>
<thead>
<tr>
<th>Interface</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ActionListener</td>
<td>Defines one method to receive action events.</td>
</tr>
<tr>
<td>AdjustmentListener</td>
<td>Defines one method to receive adjustment events.</td>
</tr>
<tr>
<td>ComponentListener</td>
<td>Defines four methods to recognize when a component is hidden, moved, resized, or shown.</td>
</tr>
<tr>
<td>ContainerListener</td>
<td>Defines two methods to recognize when a component is added to or removed from a container.</td>
</tr>
<tr>
<td>FocusListener</td>
<td>Defines two methods to recognize when a component gains or loses keyboard focus.</td>
</tr>
<tr>
<td>ItemListener</td>
<td>Defines one method to recognize when the state of an item changes.</td>
</tr>
<tr>
<td>KeyListener</td>
<td>Defines three methods to recognize when a key is pressed, released, or typed.</td>
</tr>
<tr>
<td>MouseListener</td>
<td>Defines five methods to recognize when the mouse is clicked, enters a component, exits a component, is pressed, or is released.</td>
</tr>
<tr>
<td>MouseMotionListener</td>
<td>Defines two methods to recognize when the mouse is dragged or moved.</td>
</tr>
<tr>
<td>MouseWheelListener</td>
<td>Defines one method to recognize when the mouse wheel is moved.</td>
</tr>
<tr>
<td>TextListener</td>
<td>Defines one method to recognize when a text value changes.</td>
</tr>
<tr>
<td>WindowFocusListener</td>
<td>Defines two methods to recognize when a window gains or loses input focus.</td>
</tr>
<tr>
<td>WindowListener</td>
<td>Defines seven methods to recognize when a window is activated, closed, deactivated, deiconified, iconified, opened, or quit.</td>
</tr>
</tbody>
</table>

**The ActionListener Interface**

This interface defines the `actionPerformed( )` method that is invoked when an action event occurs. Its general form is shown here:

```java
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:

```java
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:

```java
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.
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.

**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:
```java
public T getN() {
    return value;
}
public void setN(T arg) {
    value = 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:
```java
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;
}
```
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:

```java
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:

```java
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:

```java
public void addTListener(TListener eventListener)
public void addTListener(TListener eventListener)
throws java.util.TooManyListenersException
```

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:

```java
public void addTemperatureListener(TemperatureListener tl) {
    ...
}
public void removeTemperatureListener(TemperatureListener tl) {
    ...
}
```
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.

<table>
<thead>
<tr>
<th>Interface</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AppletInitializer</td>
<td>Methods in this interface are used to initialize Beans that are also applets.</td>
</tr>
<tr>
<td>BeanInfo</td>
<td>This interface allows a designer to specify information about the properties, events, and methods of a Bean.</td>
</tr>
<tr>
<td>Customizer</td>
<td>This interface allows a designer to provide a graphical user interface through which a Bean may be configured.</td>
</tr>
<tr>
<td>DesignMode</td>
<td>Methods in this interface determine if a Bean is executing in design mode.</td>
</tr>
<tr>
<td>ExceptionListener</td>
<td>A method in this interface is invoked when an exception has occurred.</td>
</tr>
<tr>
<td>PropertyChangeListener</td>
<td>A method in this interface is invoked when a bound property is changed.</td>
</tr>
<tr>
<td>PropertyEditor</td>
<td>Objects that implement this interface allow designers to change and display property values.</td>
</tr>
<tr>
<td>VetoableChangeListener</td>
<td>A method in this interface is invoked when a constrained property is changed.</td>
</tr>
<tr>
<td>Visibility</td>
<td>Methods in this interface allow a Bean to execute in environments where a graphical user interface is not available.</td>
</tr>
</tbody>
</table>

**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`.

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BeanDescriptor</td>
<td>This class provides information about a Bean. It also allows you to associate a customizer with a Bean.</td>
</tr>
<tr>
<td>Beans</td>
<td>This class is used to obtain information about a Bean.</td>
</tr>
<tr>
<td>DefaultPersistenceDelegate</td>
<td>A concrete subclass of <code>PersistenceDelegate</code>.</td>
</tr>
<tr>
<td>Encoder</td>
<td>Encodes the state of a set of Beans. Can be used to write this information to a stream.</td>
</tr>
<tr>
<td>EventHandler</td>
<td>Supports dynamic event listener creation.</td>
</tr>
<tr>
<td>EventSetDescriptor</td>
<td>Instances of this class describe an event that can be generated by a Bean.</td>
</tr>
<tr>
<td>Expression</td>
<td>Encapsulates a call to a method that returns a result.</td>
</tr>
<tr>
<td>FeatureDescriptor</td>
<td>This is the superclass of the <code>PropertyDescriptor</code>, <code>EventSetDescriptor</code>, and <code>MethodDescriptor</code> classes, among others.</td>
</tr>
<tr>
<td>Class Name</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>IndexedPropertyChangeEvent</td>
<td>A subclass of PropertyChangeEvent that represents a change to an indexed property.</td>
</tr>
<tr>
<td>IndexedPropertyDescriptor</td>
<td>Instances of this class describe an indexed property of a Bean.</td>
</tr>
<tr>
<td>IntrospectionException</td>
<td>An exception of this type is generated if a problem occurs when analyzing a Bean.</td>
</tr>
<tr>
<td>Introspector</td>
<td>This class analyzes a Bean and constructs a BeanInfo object that describes the component.</td>
</tr>
<tr>
<td>MethodDescriptor</td>
<td>Instances of this class describe a method of a Bean.</td>
</tr>
<tr>
<td>ParameterDescriptor</td>
<td>Instances of this class describe a method parameter.</td>
</tr>
<tr>
<td>PersistenceDelegate</td>
<td>Handles the state information of an object.</td>
</tr>
<tr>
<td>PropertyChangeEvent</td>
<td>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.</td>
</tr>
<tr>
<td>PropertyChangeListenerProxy</td>
<td>Extends EventListenerProxy and implements PropertyChangeListener.</td>
</tr>
<tr>
<td>PropertyChangeSupport</td>
<td>Beans that support bound properties can use this class to notify PropertyChangeListener objects.</td>
</tr>
<tr>
<td>PropertyDescriptor</td>
<td>Instances of this class describe a property of a Bean.</td>
</tr>
<tr>
<td>PropertyEditorManager</td>
<td>This class locates a PropertyEditor object for a given type.</td>
</tr>
<tr>
<td>PropertyEditorSupport</td>
<td>This class provides functionality that can be used when writing property editors.</td>
</tr>
<tr>
<td>PropertyVetoException</td>
<td>An exception of this type is generated if a change to a constrained property is vetoed.</td>
</tr>
<tr>
<td>SimpleBeanInfo</td>
<td>This class provides functionality that can be used when writing BeanInfo classes.</td>
</tr>
<tr>
<td>Statement</td>
<td>Encapsulates a call to a method.</td>
</tr>
<tr>
<td>VetoableChangeListenerProxy</td>
<td>Extends EventListenerProxy and implements VetoableChangeListener.</td>
</tr>
<tr>
<td>VetoableChangeSupport</td>
<td>Beans that support constrained properties can use this class to notify VetoableChangeListener objects.</td>
</tr>
<tr>
<td>XMLDecoder</td>
<td>Used to read a Bean from an XML document.</td>
</tr>
<tr>
<td>XMLEncoder</td>
<td>Used to write a Bean to an XML document.</td>
</tr>
</tbody>
</table>