**	Matches zero or more of any character across directories.
[*chars*]	Matches any one character in *chars*. A * or ? within *chars* will be treated as a normal character, not a wildcard. A range can be specified by use of a hyphen, such as [x-z].
{*globlist*}	Matches any one of the globs specified in a comma-separated list of globs in *globlist*.

You can specify a * or ? character, using * and \?. To specify a \, use \\. You can experiment with a glob by substituting this call to **newDirectoryStream( )** into the previous program:

```
Files.newDirectoryStream(Paths.get(dirname), "{Path,Dir}*.{java,class}")
```

This obtains a directory stream that contains only those files whose names begin with either "Path" or "Dir" and use either the "java" or "class" extension. Thus, it would match names like **DirList.java** and **PathDemo.java**, but not **MyPathDemo.java**, for example.

Another way to filter a directory is to use this version of **newDirectoryStream( )**:

```
static DirectoryStream<Path> newDirectoryStream(Path dirPath,
                                                DirectoryStream.Filter<? super Path> filefilter)
                                                  throws IOException
```

Here, **DirectoryStream.Filter** is an interface that specifies the following method:

```
boolean accept(T entry) throws IOException
```

In this case, **T** will be **Path**. If you want to include *entry* in the list, return **true**. Otherwise, return **false**. This form of **newDirectoryStream( )** offers the advantage of being able to filter a directory based on something other than a filename. For example, you can filter based on size, creation date, modification date, or attribute, to name a few.

The following program demonstrates the process. It will list only those files that are writable.

```
// Display a directory of only those files that are writable.

import java.io.*;
import java.nio.file.*;
import java.nio.file.attribute.*;

class DirList {
  public static void main(String args[]) {
    String dirname = "\\MyDir";

    // Create a filter that returns true only for writable files.
    DirectoryStream.Filter<Path> how = new DirectoryStream.Filter<Path>() {
      public boolean accept(Path filename) throws IOException {
        if(Files.isWritable(filename)) return true;
        return false;
      }
    };
```

```
    // Obtain and manage a directory stream of writable files.
    try (DirectoryStream<Path> dirstrm =
           Files.newDirectoryStream(Paths.get(dirname), how) )
    {
      System.out.println("Directory of " + dirname);

      for(Path entry : dirstrm) {
        BasicFileAttributes attribs =
          Files.readAttributes(entry, BasicFileAttributes.class);

        if(attribs.isDirectory())
          System.out.print("<DIR> ");
        else
          System.out.print("      ");

        System.out.println(entry.getName(1));
      }
    } catch(InvalidPathException e) {
      System.out.println("Path Error " + e);
    } catch(NotDirectoryException e) {
      System.out.println(dirname + " is not a directory.");
    } catch (IOException e) {
      System.out.println("I/O Error: " + e);
    }
  }
}
```

Use walkFileTree( ) to List a Directory Tree

The preceding examples have obtained the contents of only a single directory. However, sometimes you will want to obtain a list of the files in a directory tree. In the past, this was quite a chore, but NIO.2 makes it easy because now you can use the **walkFileTree( )** method defined by **Files** to process a directory tree. It has two forms. The one used in this chapter is shown here:

```
static Path walkFileTree(Path root, FileVisitor<? extends Path> fv)
              throws IOException
```

The path to the starting point of the directory walk is passed in *root*. An instance of **FileVisitor** is passed in *fv*. The implementation of **FileVisitor** determines how the directory tree is traversed, and it gives you access to the directory information. If an I/O error occurs, an **IOException** is thrown. A **SecurityException** is also possible.

FileVisitor is an interface that defines how files are visited when a directory tree is traversed. It is a generic interface that is declared like this:

```
interface FileVisitor<T>
```

For use in **walkFileTree( )**, **T** will be **Path** (or any type derived from **Path**). **FileVisitor** defines the following methods.

Method	Description
FileVisitResult postVisitDirectory(T *dir*, IOException *exc*) throws IOException	Called after a directory has been visited. The directory is passed in *dir*, and any **IOException** is passed in *exc*. If *exc* is **null**, no exception occurred. The result is returned.
FileVisitResult preVisitDirectory(T *dir*, BasicFileAttributes *attribs*) throws IOException	Called before a directory is visited. The directory is passed in *dir*, and the attributes associated with the directory are passed in *attribs*. The result is returned. To examine the directory, return **FileVisitResult.CONTINUE**.
FileVisitResult visitFile(T *file*, BasicFileAttributes *attribs*) throws IOException	Called when a file is visited. The file is passed in *file*, and the attributes associated with the file are passed in *attribs*. The result is returned.
FileVisitResult visitFileFailed(T *file*, IOException *exc*) throws IOException	Called when an attempt to visit a file fails. The file that failed is passed in *file*, and the **IOException** is passed in *exc*. The result is returned.

Notice that each method returns a **FileVisitResult**. This enumeration defines the following values:

CONTINUE	SKIP_SIBLINGS	SKIP_SUBTREE	TERMINATE

In general, to continue traversing the directory and subdirectories, a method should return **CONTINUE**. For **preVisitDirectory( )**, return **SKIP_SIBLINGS** to bypass the directory and its siblings and prevent **postVisitDirectory( )** from being called. To bypass just the directory and subdirectories, return **SKIP_SUBTREE**. To stop the directory traversal, return **TERMINATE**.

Although it is certainly possible to create your own visitor class that implements these methods defined by **FileVisitor**, you won't normally do so because a simple implementation is provided by **SimpleFileVisitor**. You can just override the default implementation of the method or methods in which you are interested. Here is a short example that illustrates the process. It displays all files in the directory tree that has **\MyDir** as its root. Notice how short this program is.

```
// A simple example that uses walkFileTree( ) to display a directory tree.
// Requires JDK 7 or later.

import java.io.*;
import java.nio.file.*;
import java.nio.file.attribute.*;


// Create a custom version of SimpleFileVisitor that overrides
// the visitFile( ) method.
class MyFileVisitor extends SimpleFileVisitor<Path> {
  public FileVisitResult visitFile(Path path, BasicFileAttributes attribs)
```

```
    throws IOException
  {
    System.out.println(path);
    return FileVisitResult.CONTINUE;
  }
}

class DirTreeList {
  public static void main(String args[]) {
    String dirname = "\\MyDir";

    System.out.println("Directory tree starting with " + dirname + ":\n");

    try {
      Files.walkFileTree(Paths.get(dirname), new MyFileVisitor());
    } catch (IOException exc) {
      System.out.println("I/O Error");
    }
  }
}
```

Here is sample output produced by the program when used on the same **MyDir** directory shown earlier. In this example, the subdirectory called **examples** contains one file called **MyProgram.java**.

```
Directory tree starting with \MyDir:

\MyDir\DirList.class
\MyDir\DirList.java
\MyDir\examples\MyProgram.java
\MyDir\Test.txt
```

In the program, the class **MyFileVisitor** extends **SimpleFileVisitor**, overriding only the **visitFile( )** method. In this example, **visitFile( )** simply displays the files, but more sophisticated functionality is easy to achieve. For example, you could filter the files or perform actions on the files, such as copying them to a backup device. For the sake of clarity, a named class was used to override **visitFile( )**, but you could also use an anonymous inner class.

One last point: It is possible to watch a directory for changes by using **java.nio.file.WatchService**.

Pre-JDK 7 Channel-Based Examples

Before concluding this chapter, one more aspect of NIO needs to be covered. The preceding sections have used several of the new features added to NIO by JDK 7. However, you may encounter pre-JDK 7 code that will need to be maintained or possibly converted to use the new features. For this reason, the following sections show how to read and write files using the pre-JDK 7 NIO system. They do so by reworking some of the examples shown earlier so that they use the original NIO features, rather than those supported by NIO.2. This means that the examples in this section will work with versions of Java prior to JDK 7.

The key difference between pre-JDK 7 NIO code and newer NIO code is the **Path** interface, which was added by JDK 7. Thus, pre-JDK 7 code does not use **Path** to describe a file or open a channel to it. Also, pre-JDK 7 code does not use **try**-with-resource statements since automatic resource management was also added by JDK 7.

REMEMBER The examples in this section describe how legacy NIO code works. This section is strictly for the benefit of those programmers working on pre-JDK 7 code or using pre-JDK 7 compilers. New code should take advantage of the NIO features added by JDK 7.

Read a File, Pre-JDK 7

This section reworks the two channel-based file input examples shown earlier so they use only pre-JDK 7 features. The first example reads a file by manually allocating a buffer and then performing an explicit read operation. The second example uses a mapped file, which automates the process.

When using a pre-JDK 7 version of Java to read a file using a channel and a manually allocated buffer, you first open the file for input using **FileInputStream**, using the same mechanism explained in Chapter 20. Next, obtain a channel to this file by calling **getChannel( )** on the **FileInputStream** object. It has this general form:

```
FileChannel getChannel( )
```

It returns a **FileChannel** object, which encapsulates the channel for file operations. Then, call **allocate( )** to allocate a buffer. Because file channels operate on byte buffers, you will use the **allocate( )** method defined by **ByteBuffer**, which works as previously described.

The following program shows how to read and display a file called **test.txt** through a channel using explicit input operations for versions of Java prior to JDK 7:

```
// Use Channels to read a file. Pre-JDK 7 version.
import java.io.*;
import java.nio.*;
import java.nio.channels.*;

public class ExplicitChannelRead {
  public static void main(String args[]) {
    FileInputStream fIn = null;
    FileChannel fChan = null;
    ByteBuffer mBuf;
    int count;

    try {
      // First, open a file for input.
      fIn = new FileInputStream("test.txt");

      // Next, obtain a channel to that file.
      fChan = fIn.getChannel();

      // Allocate a buffer.
      mBuf = ByteBuffer.allocate(128);

      do {
```

```
        // Read a buffer.
        count = fChan.read(mBuf);

        // Stop when end of file is reached.
        if(count != -1) {

          // Rewind the buffer so that it can be read.
          mBuf.rewind();

          // Read bytes from the buffer and show
          // them on the screen.
          for(int i=0; i < count; i++)
            System.out.print((char)mBuf.get());
        }
      } while(count != -1);

      System.out.println();

    } catch (IOException e) {
      System.out.println("I/O Error " + e);
    } finally {
      try {
        if(fChan != null) fChan.close(); // close channel
      } catch(IOException e) {
        System.out.println("Error Closing Channel.");
      }
      try {
        if(fIn != null) fIn.close(); // close file
      } catch(IOException e) {
        System.out.println("Error Closing File.");
      }
    }
  }
}
```

In this program, notice that the file is opened by using the **FileInputStream** constructor, and a reference to that object is assigned to **fIn**. Next, a channel connected to the file is obtained by calling **getChannel( )** on **fIn**. After this point, the program works like the NIO.2 version shown previously. To synopsize: The program then calls the **allocate( )** method of **ByteBuffer** to allocate a buffer that will hold the contents of the file when it is read. A byte buffer is used because **FileChannel** operates on bytes. A reference to this buffer is stored in **mBuf**. The contents of the file are then read, one buffer at a time, into **mBuf** through a call to **read( )**. The number of bytes read is stored in **count**. Next, the buffer is rewound through a call to **rewind( )**. This call is necessary because the current position is at the end of the buffer after the call to **read( )**, and it must be reset to the start of the buffer in order for the bytes in **mBuf** to be read by calling **get( )**. When the end of the file has been reached, the value returned by **read( )** will be –1. When this occurs, the program ends, explicitly closing the channel and the file.

Another way to read a file is to map it to a buffer. As explained earlier, a principal advantage to this approach is that the buffer automatically contains the contents of the file. No explicit read operation is necessary. To map and read the contents of a file using

pre-JDK 7 NIO, first open the file using **FileInputStream**. Next, obtain a channel to that file by calling **getChannel( )** on the file object. Then, map the channel to a buffer by calling **map( )** on the **FileChannel** object. The **map( )** method works as described earlier.

The following program reworks the preceding example so that it uses only pre-JDK 7 features to create a mapped file:

```
// Use a mapped file to read a file. Pre-JDK 7 version.

import java.io.*;
import java.nio.*;
import java.nio.channels.*;

public class MappedChannelRead {
  public static void main(String args[]) {
    FileInputStream fIn = null;
    FileChannel fChan = null;
    long fSize;
    MappedByteBuffer mBuf;

    try {
      // First, open a file for input.
      fIn = new FileInputStream("test.txt");

      // Next, obtain a channel to that file.
      fChan = fIn.getChannel();

      // Get the size of the file.
      fSize = fChan.size();

      // Now, map the file into a buffer.
      mBuf = fChan.map(FileChannel.MapMode.READ_ONLY, 0, fSize);

      // Read and display bytes from buffer.
      for(int i=0; i < fSize; i++)
        System.out.print((char)mBuf.get());

    } catch (IOException e) {
      System.out.println("I/O Error " + e);
    } finally {
      try {
        if(fChan != null) fChan.close(); // close channel
      } catch(IOException e) {
        System.out.println("Error Closing Channel.");
      }
      try {
        if(fIn != null) fIn.close(); // close file
      } catch(IOException e) {
        System.out.println("Error Closing File.");
      }
    }
  }
}
```

In the program, the file is opened by using the **FileInputStream** constructor, and a reference to that object is assigned to **fIn**. A channel connected to the file is obtained by calling **getChannel( )** on **fIn**. Next, the size of the file is obtained. Then, the entire file is mapped into memory by calling **map( )**, and a reference to the buffer is stored in **mBuf**. The bytes in **mBuf** are read by calling **get( )**.

Write to a File, Pre-JDK 7

This section reworks the two channel-based file output examples shown earlier so that they use only pre-JDK 7 features. The first example writes to a file by manually allocating a buffer and then performing an explicit output operation. The second example uses a mapped file, which automates the process. In both cases, neither **Path** nor **try**-with-resources is used. This is because neither were part of Java until JDK 7.

When using a pre-JDK 7 version of Java to write a file using a channel and a manually allocated buffer, first open the file for output. This is done by creating a **FileOutputStream**, as described in Chapter 20. Next, obtain a channel to the file by calling **getChannel( )** and then allocate a byte buffer by calling **allocate( )**, as described in the previous section. Next, put the data you want to write into that buffer, and then call **write( )** on the channel. The following program demonstrates this procedure. It writes the alphabet to a file called **test.txt**.

```
// Write to a file using NIO. Pre-JDK 7 Version.
import java.io.*;
import java.nio.*;
import java.nio.channels.*;

public class ExplicitChannelWrite {
  public static void main(String args[]) {
    FileOutputStream fOut = null;
    FileChannel fChan = null;
    ByteBuffer mBuf;

    try {
      // First, open the output file.
      fOut = new FileOutputStream("test.txt");

      // Next, get a channel to the output file.
      fChan = fOut.getChannel();

      // Create a buffer.
      mBuf = ByteBuffer.allocate(26);

      // Write some bytes to the buffer.
      for(int i=0; i<26; i++)
        mBuf.put((byte)('A' + i));

      // Rewind the buffer so that it can be written.
      mBuf.rewind();

      // Write the buffer to the output file.
      fChan.write(mBuf);
```

```
    } catch (IOException e) {
      System.out.println("I/O Error " + e);
    } finally {
      try {
        if(fChan != null) fChan.close(); // close channel
      } catch(IOException e) {
        System.out.println("Error Closing Channel.");
      }
      try {
        if(fOut != null) fOut.close(); // close file
      } catch(IOException e) {
        System.out.println("Error Closing File.");
      }
    }
  }
}
```

The call to **rewind( )** on **mBuf** is necessary in order to reset the current position to zero after data has been written to **mBuf**. Remember, each call to **put( )** advances the current position. Therefore, it is necessary for the current position to be reset to the start of the buffer before calling **write( )**. If this is not done, **write( )** will think that there is no data in the buffer.

When using a pre-JDK 7 version of Java to write to a file using a mapped file, follow these steps. First, open the file for read/write operations by creating a **RandomAccessFile** object. This is necessary to enable the file to be both read from and written to. Next, map that file to a buffer by calling **map( )** on that object. Then, write to the buffer. Because the buffer is mapped to the file, any changes to that buffer are automatically reflected in the file. Thus, no explicit write operations to the channel are necessary.

Here is the preceding program reworked so that a mapped file is used:

```
// Write to a mapped file. Pre JDK 7 version.
import java.io.*;
import java.nio.*;
import java.nio.channels.*;

public class MappedChannelWrite {
  public static void main(String args[]) {
    RandomAccessFile fOut = null;
    FileChannel fChan = null;
    ByteBuffer mBuf;

    try {
      fOut = new RandomAccessFile("test.txt", "rw");

      // Next, obtain a channel to that file.
      fChan = fOut.getChannel();

      // Then, map the file into a buffer.
      mBuf = fChan.map(FileChannel.MapMode.READ_WRITE, 0, 26);
```

```
      // Write some bytes to the buffer.
      for(int i=0; i<26; i++)
        mBuf.put((byte)('A' + i));

    } catch (IOException e) {
      System.out.println("I/O Error " + e);
    } finally {
      try {
        if(fChan != null) fChan.close(); // close channel
      } catch(IOException e) {
        System.out.println("Error Closing Channel.");
      }
      try {
        if(fOut != null) fOut.close(); // close file
      } catch(IOException e) {
        System.out.println("Error Closing File.");
      }
    }
  }
}
```

As you can see, there are no explicit write operations to the channel itself. Because **mBuf** is mapped to the file, changes to **mBuf** are automatically reflected in the underlying file.

This page has been intentionally left blank

CHAPTER

22 Networking

As all readers know, Java is practically a synonym for Internet programming. There are a number of reasons for this, not the least of which is its ability to generate secure, cross-platform, portable code. However, one of the most important reasons that Java is the premier language for network programming are the classes defined in the **java.net** package. They provide an easy-to-use means by which programmers of all skill levels can access network resources.

This chapter explores the **java.net** package. It is important to emphasize that networking is a very large and at times complicated topic. It is not possible for this book to discuss all of the capabilities contained in **java.net.** Instead, this chapter focuses on several of its core classes and interfaces.

Networking Basics

Before we begin, it will be useful to review some key networking concepts and terms. At the core of Java's networking support is the concept of a *socket*. A socket identifies an endpoint in a network. The socket paradigm was part of the 4.2BSD Berkeley UNIX release in the early 1980s. Because of this, the term *Berkeley socket* is also used. Sockets are at the foundation of modern networking because a socket allows a single computer to serve many different clients at once, as well as to serve many different types of information. This is accomplished through the use of a *port*, which is a numbered socket on a particular machine. A server process is said to "listen" to a port until a client connects to it. A server is allowed to accept multiple clients connected to the same port number, although each session is unique. To manage multiple client connections, a server process must be multithreaded or have some other means of multiplexing the simultaneous I/O.

Socket communication takes place via a protocol. *Internet Protocol (IP)* is a low-level routing protocol that breaks data into small packets and sends them to an address across a network, which does not guarantee to deliver said packets to the destination. *Transmission Control Protocol* (TCP) is a higher-level protocol that manages to robustly string together these packets, sorting and retransmitting them as necessary to reliably transmit data. A third protocol, *User Datagram Protocol (UDP),* sits next to TCP and can be used directly to support fast, connectionless, unreliable transport of packets.

Once a connection has been established, a higher-level protocol ensues, which is dependent on which port you are using. TCP/IP reserves the lower 1,024 ports for specific protocols. Many of these will seem familiar to you if you have spent any time surfing the Internet. Port number 21 is for FTP; 23 is for Telnet; 25 is for e-mail; 43 is for whois; 80 is for HTTP; 119 is for netnews—and the list goes on. It is up to each protocol to determine how a client should interact with the port.

For example, HTTP is the protocol that web browsers and servers use to transfer hypertext pages and images. It is a quite simple protocol for a basic page-browsing web server. Here's how it works. When a client requests a file from an HTTP server, an action known as a *hit*, it simply sends the name of the file in a special format to a predefined port and reads back the contents of the file. The server also responds with a status code to tell the client whether or not the request can be fulfilled and why.

A key component of the Internet is the *address*. Every computer on the Internet has one. An Internet address is a number that uniquely identifies each computer on the Net. Originally, all Internet addresses consisted of 32-bit values, organized as four 8-bit values. This address type was specified by IPv4 (Internet Protocol, version 4). However, a new addressing scheme, called IPv6 (Internet Protocol, version 6) has come into play. IPv6 uses a 128-bit value to represent an address, organized into eight 16-bit chunks. Although there are several reasons for and advantages to IPv6, the main one is that it supports a much larger address space than does IPv4. Fortunately, when using Java, you won't normally need to worry about whether IPv4 or IPv6 addresses are used because Java handles the details for you.

Just as the numbers of an IP address describe a network hierarchy, the name of an Internet address, called its *domain name*, describes a machine's location in a name space. For example, **www.HerbSchildt.com** is in the *COM* top-level domain (reserved for U.S. commercial sites); it is called *HerbSchildt*, and *www* identifies the server for web requests. An Internet domain name is mapped to an IP address by the *Domain Naming Service (DNS)*. This enables users to work with domain names, but the Internet operates on IP addresses.

The Networking Classes and Interfaces

Java supports TCP/IP both by extending the already established stream I/O interface introduced in Chapter 20 and by adding the features required to build I/O objects across the network. Java supports both the TCP and UDP protocol families. TCP is used for reliable stream-based I/O across the network. UDP supports a simpler, hence faster, point-to-point datagram-oriented model. The classes contained in the **java.net** package are shown here:

Authenticator	InetAddress	SocketAddress
CacheRequest	InetSocketAddress	SocketImpl
CacheResponse	InterfaceAddress	SocketPermission
ContentHandler	JarURLConnection	StandardSocketOption
CookieHandler	MulticastSocket	URI
CookieManager	NetPermission	URL
DatagramPacket	NetworkInterface	URLClassLoader

DatagramSocket	PasswordAuthentication	URLConnection
DatagramSocketImpl	Proxy	URLDecoder
HttpCookie	ProxySelector	URLEncoder
HttpURLConnection	ResponseCache	URLPermission (Added by JDK 8.)
IDN	SecureCacheResponse	URLStreamHandler
Inet4Address	ServerSocket	
Inet6Address	Socket	

The **java.net** package's interfaces are listed here:

ContentHandlerFactory	FileNameMap	SocketOptions
CookiePolicy	ProtocolFamily	URLStreamHandlerFactory
CookieStore	SocketImplFactory	
DatagramSocketImplFactory	SocketOption	

In the sections that follow, we will examine the main networking classes and show several examples that apply to them. Once you understand these core networking classes, you will be able to easily explore the others on your own.

InetAddress

The **InetAddress** class is used to encapsulate both the numerical IP address and the domain name for that address. You interact with this class by using the name of an IP host, which is more convenient and understandable than its IP address. The **InetAddress** class hides the number inside. **InetAddress** can handle both IPv4 and IPv6 addresses.

Factory Methods

The **InetAddress** class has no visible constructors. To create an **InetAddress** object, you have to use one of the available factory methods. *Factory methods* are merely a convention whereby static methods in a class return an instance of that class. This is done in lieu of overloading a constructor with various parameter lists when having unique method names makes the results much clearer. Three commonly used **InetAddress** factory methods are shown here:

```
static InetAddress getLocalHost( )
  throws UnknownHostException

static InetAddress getByName(String hostName)
  throws UnknownHostException

static InetAddress[ ] getAllByName(String hostName)
  throws UnknownHostException
```

The **getLocalHost( )** method simply returns the **InetAddress** object that represents the local host. The **getByName( )** method returns an **InetAddress** for a host name passed to it. If these methods are unable to resolve the host name, they throw an **UnknownHostException**.

On the Internet, it is common for a single name to be used to represent several machines. In the world of web servers, this is one way to provide some degree of scaling. The **getAllByName( )** factory method returns an array of **InetAddress**es that represent all of the addresses that a particular name resolves to. It will also throw an **UnknownHostException** if it can't resolve the name to at least one address.

InetAddress also includes the factory method **getByAddress( ),** which takes an IP address and returns an **InetAddress** object. Either an IPv4 or an IPv6 address can be used.

The following example prints the addresses and names of the local machine and two Internet web sites:

```
// Demonstrate InetAddress.
import java.net.*;

class InetAddressTest
{
  public static void main(String args[]) throws UnknownHostException {
    InetAddress Address = InetAddress.getLocalHost();
    System.out.println(Address);

    Address = InetAddress.getByName("www.HerbSchildt.com");
    System.out.println(Address);

    InetAddress SW[] = InetAddress.getAllByName("www.nba.com");
    for (int i=0; i<SW.length; i++)
      System.out.println(SW[i]);
  }
}
```

Here is the output produced by this program. (Of course, the output you see may be slightly different.)

```
default/166.203.115.212
www.HerbSchildt.com/216.92.65.4
www.nba.com/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.

NOTE As a general rule, applets may only establish socket connections back to the host from which the applet was downloaded. This restriction exists because it would be dangerous for applets loaded through a firewall to have access to any arbitrary machine.

There are two kinds of TCP sockets in Java. One is for servers, and the other is for clients. The **ServerSocket** class is designed to be a "listener," which waits for clients to connect before doing anything. Thus, **ServerSocket** is for servers. The **Socket** class is for clients. It is designed to connect to server sockets and initiate protocol exchanges. Because client sockets are the most commonly used by Java applications, they are examined here.

The creation of a **Socket** object implicitly establishes a connection between the client and server. There are no methods or constructors that explicitly expose the details of establishing that connection. Here are two constructors used to create client sockets:

Socket(String *hostName*, int *port*) throws UnknownHostException, IOException	Creates a socket connected to the named host and port.
Socket(InetAddress *ipAddress*, int *port*) throws IOException	Creates a socket using a preexisting **InetAddress** object and a port.

Socket defines several instance methods. For example, a **Socket** can be examined at any time for the address and port information associated with it, by use of the following methods:

InetAddress getInetAddress( )	Returns the **InetAddress** associated with the **Socket** object. It returns **null** if the socket is not connected.
int getPort( )	Returns the remote port to which the invoking **Socket** object is connected. It returns 0 if the socket is not connected.
int getLocalPort( )	Returns the local port to which the invoking **Socket** object is bound. It returns –1 if the socket is not bound.

You can gain access to the input and output streams associated with a **Socket** by use of the **getInputStream( )** and **getOuptutStream( )** methods, as shown here. Each can throw an **IOException** if the socket has been invalidated by a loss of connection. These streams are used exactly like the I/O streams described in Chapter 20 to send and receive data.

InputStream getInputStream( ) throws IOException	Returns the **InputStream** associated with the invoking socket.
OutputStream getOutputStream( ) throws IOException	Returns the **OutputStream** associated with the invoking socket.

Several other methods are available, including **connect( ),** which allows you to specify a new connection; **isConnected( ),** which returns true if the socket is connected to a server; **isBound( ),** which returns true if the socket is bound to an address; and **isClosed( ),** which returns true if the socket is closed. To close a socket, call **close( )**. Closing a socket also closes the I/O streams associated with the socket. Beginning with JDK 7, **Socket** also implements **AutoCloseable**, which means that you can use a **try**-with-resources block to manage a socket.

The following program provides a simple **Socket** example. It opens a connection to a "whois" port (port 43) on the InterNIC server, sends the command-line argument down the socket, and then prints the data that is returned. InterNIC will try to look up the argument as a registered Internet domain name, and then send back the IP address and contact information for that site.

```
// Demonstrate Sockets.
import java.net.*;
import java.io.*;

class Whois {
  public static void main(String args[]) throws Exception {
    int c;

    // Create a socket connected to internic.net, port 43.
    Socket s = new Socket("whois.internic.net", 43);

    // Obtain input and output streams.
    InputStream in = s.getInputStream();
    OutputStream out = s.getOutputStream();

    // Construct a request string.

    String str = (args.length == 0 ? "MHProfessional.com" : args[0]) + "\n";
    // Convert to bytes.
    byte buf[] = str.getBytes();

    // Send request.
    out.write(buf);

    // Read and display response.
    while ((c = in.read()) != -1) {
      System.out.print((char) c);
    }
    s.close();
  }
}
```

If, for example, you obtained information about **MHProfessional.com**, you'd get something similar to the following:

```
Whois Server Version 2.0

Domain names in the .com and .net domains can now be registered
with many different competing registrars. Go to http://www.internic.net
for detailed information.

   Domain Name: MHPROFESSIONAL.COM
   Registrar: CSC CORPORATE DOMAINS, INC.
   Whois Server: whois.corporatedomains.com
   Referral URL: http://www.cscglobal.com
   Name Server: NS1.MHEDU.COM
   Name Server: NS2.MHEDU.COM
.
.
.
```

Here is how the program works. First, a **Socket** is constructed that specifies the host name "whois.internic.net" and the port number 43. **Internic.net** is the InterNIC web site that handles whois requests. Port 43 is the whois port. Next, both input and output streams are opened on the socket. Then, a string is constructed that contains the name of the web site you want to obtain information about. In this case, if no web site is specified on the command line, then "MHProfessional.com" is used. The string is converted into a **byte** array and then sent out of the socket. The response is read by inputting from the socket, and the results are displayed. Finally, the socket is closed, which also closes the I/O streams.

In the preceding example, the socket was closed manually by calling **close( )**. If you are using JDK 7 or later, then you can use a **try**-with-resources block to automatically close the socket. For example, here is another way to write the **main( )** method of the previous program:

```
// Use try-with-resources to close a socket.
public static void main(String args[]) throws Exception {
  int c;

  // Create a socket connected to internic.net, port 43. Manage this
  // socket with a try-with-resources block.
  try ( Socket s = new Socket("whois.internic.net", 43) ) {

    // Obtain input and output streams.
    InputStream in = s.getInputStream();
    OutputStream out = s.getOutputStream();

    // Construct a request string.
    String str = (args.length == 0 ? "MHProfessional.com" : args[0]) + "\n";
    // Convert to bytes.
    byte buf[] = str.getBytes();

    // Send request.
    out.write(buf);

    // Read and display response.
    while ((c = in.read()) != -1) {
      System.out.print((char) c);
    }
  }
  // The socket is now closed.
}
```

In this version, the socket is automatically closed when the **try** block ends.

So the examples will work with earlier versions of Java and to clearly illustrate when a network resource can be closed, subsequent examples will continue to call **close( )** explicitly. However, in your own code, you should consider using automatic resource management since it offers a more streamlined approach. One other point: In this version, exceptions are still thrown out of **main( )**, but they could be handled by adding **catch** clauses to the end of the **try**-with-resources block.

NOTE For simplicity, the examples in this chapter simply throw all exceptions out of **main( )**. This allows the logic of the network code to be clearly illustrated. However, in real-world code, you will normally need to handle the exceptions in an appropriate way.

URL

The preceding example was rather obscure because the modern Internet is not about the older protocols such as whois, finger, and FTP. It is about WWW, the World Wide Web. The Web is a loose collection of higher-level protocols and file formats, all unified in a web browser. One of the most important aspects of the Web is that Tim Berners-Lee devised a scalable way to locate all of the resources of the Net. Once you can reliably name anything and everything, it becomes a very powerful paradigm. The Uniform Resource Locator (URL) does exactly that.

The URL provides a reasonably intelligible form to uniquely identify or address information on the Internet. URLs are ubiquitous; every browser uses them to identify information on the Web. Within Java's network class library, the **URL** class provides a simple, concise API to access information across the Internet using URLs.

All URLs share the same basic format, although some variation is allowed. Here are two examples: **http://www.MHProfessional.com/** and **http://www.MHProfessional.com:80/index.htm**. A URL specification is based on four components. The first is the protocol to use, separated from the rest of the locator by a colon (:). Common protocols are HTTP, FTP, gopher, and file, although these days almost everything is being done via HTTP (in fact, most browsers will proceed correctly if you leave off the "http://" from your URL specification). The second component is the host name or IP address of the host to use; this is delimited on the left by double slashes (//) and on the right by a slash (/) or optionally a colon (:). The third component, the port number, is an optional parameter, delimited on the left from the host name by a colon (:) and on the right by a slash (/). (It defaults to port 80, the predefined HTTP port; thus, ":80" is redundant.) The fourth part is the actual file path. Most HTTP servers will append a file named **index.html** or **index.htm** to URLs that refer directly to a directory resource. Thus, **http://www.MHProfessional.com/** is the same as **http://www.MHProfessional.com/index.htm**.

Java's **URL** class has several constructors; each can throw a **MalformedURLException**. One commonly used form specifies the URL with a string that is identical to what you see displayed in a browser:

URL(String *urlSpecifier*) throws MalformedURLException

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

URL(String *protocolName*, String *hostName*, int *port*, String *path*)
throws MalformedURLException

URL(String *protocolName*, String *hostName*, String *path*)
throws MalformedURLException

Another frequently used constructor allows you to use an existing URL as a reference context and then create a new URL from that context. Although this sounds a little contorted, it's really quite easy and useful.

URL(URL *urlObj*, String *urlSpecifier*) throws MalformedURLException

The following example creates a URL to a page on **HerbSchildt.com** and then examines its properties:

```
// Demonstrate URL.
import java.net.*;
class URLDemo {
  public static void main(String args[]) throws MalformedURLException {
    URL hp = new URL(http://www.HerbSchildt.com/WhatsNew");

    System.out.println("Protocol: " + hp.getProtocol());
    System.out.println("Port: " + hp.getPort());

    System.out.println("Host: " + hp.getHost());
    System.out.println("File: " + hp.getFile());
    System.out.println("Ext:" + hp.toExternalForm());
  }
}
```

When you run this, you will get the following output:

```
Protocol: http
Port: -1
Host:  www.HerbSchildt.com
File: /WhatsNew
Ext:http://www.HerbSchildt.com/WhatsNew
```

Notice that the port is –1; this means that a port was not explicitly set. Given a **URL** object, you can retrieve the data associated with it. To access the actual bits or content information of a **URL,** create a **URLConnection** object from it, using its **openConnection( )** method, like this:

```
urlc = url.openConnection()
```

openConnection( ) has the following general form:

URLConnection openConnection( ) throws IOException

It returns a **URLConnection** object associated with the invoking **URL** object. Notice that it may throw an **IOException**.

URLConnection

URLConnection is a general-purpose class for accessing the attributes of a remote resource. Once you make a connection to a remote server, you can use **URLConnection** to inspect the properties of the remote object before actually transporting it locally. These attributes

are exposed by the HTTP protocol specification and, as such, only make sense for **URL** objects that are using the HTTP protocol.

URLConnection defines several methods. Here is a sampling:

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

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

The following example creates a **URLConnection** using the **openConnection( )** method of a **URL** object and then uses it to examine the document's properties and content:

```
// Demonstrate URLConnection.
import java.net.*;
import java.io.*;
import java.util.Date;

class UCDemo
{
  public static void main(String args[]) throws Exception {
    int c;
    URL hp = new URL(http://www.internic.net");
    URLConnection hpCon = hp.openConnection();

    // get date
    long d = hpCon.getDate();
    if(d==0)
      System.out.println("No date information.");
    else
      System.out.println("Date: " + new Date(d));

    // get content type
    System.out.println("Content-Type: " + hpCon.getContentType());

    // get expiration date
    d = hpCon.getExpiration();
    if(d==0)
      System.out.println("No expiration information.");
    else
      System.out.println("Expires: " + new Date(d));

    // get last-modified date
    d = hpCon.getLastModified();
    if(d==0)
      System.out.println("No last-modified information.");
    else
      System.out.println("Last-Modified: " + new Date(d));

    // get content length
    long len = hpCon.getContentLengthLong();
    if(len == -1)
      System.out.println("Content length unavailable.");
    else
      System.out.println("Content-Length: " + len);

    if(len != 0) {
      System.out.println("=== Content ===");
      InputStream input = hpCon.getInputStream();
      while (((c = input.read()) != -1)) {
        System.out.print((char) c);
      }
      input.close();
```

```
    } else {
      System.out.println("No content available.");
    }
  }
}
```

The program establishes an HTTP connection to **www.internic.net** over port 80. It then displays several header values and retrieves the content. You might find it interesting to try this example, observing the results, and then for comparison purposes try a different web site of your own choosing.

HttpURLConnection

Java provides a subclass of **URLConnection** that provides support for HTTP connections. This class is called **HttpURLConnection.** You obtain an **HttpURLConnection** in the same way just shown, by calling **openConnection( )** on a **URL** object, but you must cast the result to **HttpURLConnection.** (Of course, you must make sure that you are actually opening an HTTP connection.) Once you have obtained a reference to an **HttpURLConnection** object, you can use any of the methods inherited from **URLConnection.** You can also use any of the several methods defined by **HttpURLConnection.** Here is a sampling:

static boolean getFollowRedirects( )	Returns **true** if redirects are automatically followed and **false** otherwise. This feature is on by default.
String getRequestMethod( )	Returns a string representing how URL requests are made. The default is GET. Other options, such as POST, are available.
int getResponseCode( ) throws IOException	Returns the HTTP response code. –1 is returned if no response code can be obtained. An **IOException** is thrown if the connection fails.
String getResponseMessage( ) throws IOException	Returns the response message associated with the response code. Returns **null** if no message is available. An **IOException** is thrown if the connection fails.
static void setFollowRedirects(boolean *how*)	If *how* is **true,** then redirects are automatically followed. If *how* is **false,** redirects are not automatically followed. By default, redirects are automatically followed.
void setRequestMethod(String *how*) throws ProtocolException	Sets the method by which HTTP requests are made to that specified by *how*. The default method is GET, but other options, such as POST, are available. If *how* is invalid, a **ProtocolException** is thrown.

The following program demonstrates **HttpURLConnection.** It first establishes a connection to **www.google.com**. Then it displays the request method, the response code, and the response message. Finally, it displays the keys and values in the response header.

```
// Demonstrate HttpURLConnection.
import java.net.*;
import java.io.*;
import java.util.*;

class HttpURLDemo
{

public static void main(String args[]) throws Exception {
  URL hp = new URL(http://www.google.com");

  HttpURLConnection hpCon = (HttpURLConnection) hp.openConnection();

  // Display request method.
  System.out.println("Request method is " +
                     hpCon.getRequestMethod());

  // Display response code.
  System.out.println("Response code is " +
                     hpCon.getResponseCode());

  // Display response message.
  System.out.println("Response Message is " +
                     hpCon.getResponseMessage());

  // Get a list of the header fields and a set
  // of the header keys.
  Map<String, List<String>> hdrMap = hpCon.getHeaderFields();
  Set<String> hdrField = hdrMap.keySet();

  System.out.println("\nHere is the header:");

   // Display all header keys and values.
   for(String k : hdrField) {
     System.out.println("Key: " + k +
                        "  Value: " + hdrMap.get(k));
   }
  }
}
```

The output produced by the program is shown here. (Of course, the exact response returned by **www.google.com** will vary over time.)

```
Request method is GET
Response code is 200
Response Message is OK

Here is the header:
Key: Transfer-Encoding  Value: [chunked]
```

```
Key: X-Frame-Options  Value: [SAMEORIGIN]
Key: null  Value: [HTTP/1.1 200 OK]
Key: Server  Value: [gws]
Key: Cache-Control  Value: [private, max-age=0]
Key: Set-Cookie  Value:
[NID=67=rMTQWvn5eVIYA2d8F5Iu_8L-68wiMACyaXYqeSe1bvR8SzQQ_PaDCy5mNbxuw5XtdcjY
KIwmy3oVJM1Y0qZdibBOkQfJmtHpAtO61GVwumQ1ApgSXWjZ67yHxQX3g3-h; expires=Wed,
23-Apr-2014 18:31:09 GMT; path=/; domain=.google.com; HttpOnly,
PREF=ID=463b5df7b9ced9d8:FF=0:TM=1382466669:LM=1382466669:S=3LI-oT-Dzi46U1On
; expires=Thu, 22-Oct-2015 18:31:09 GMT; path=/; domain=.google.com]
Key: Expires  Value: [-1]
Key: X-XSS-Protection  Value: [1; mode=block]
Key: P3P  Value: [CP="This is not a P3P policy! See
http://www.google.com/support/accounts/bin/answer.py?hl=en&answer=151657 for
more info."]
Key: Date  Value: [Tue, 22 Oct 2013 18:31:09 GMT]
Key: Content-Type  Value: [text/html;charset=ISO-8859-1]
```

Notice how the header keys and values are displayed. First, a map of the header keys and values is obtained by calling **getHeaderFields( )** (which is inherited from **URLConnection**). Next, a set of the header keys is retrieved by calling **keySet( )** on the map. Then, the key set is cycled through by using a for-each style **for** loop. The value associated with each key is obtained by calling **get( )** on the map.

The URI Class

The **URI** class encapsulates a *Uniform Resource Identifier (URI)*. URIs are similar to URLs. In fact, URLs constitute a subset of URIs. A URI represents a standard way to identify a resource. A URL also describes how to access the resource.

Cookies

The **java.net** package includes classes and interfaces that help manage cookies and can be used to create a stateful (as opposed to stateless) HTTP session. The classes are **CookieHandler, CookieManager,** and **HttpCookie.** The interfaces are **CookiePolicy** and **CookieStore.** The creation of a stateful HTTP session is beyond the scope of this book.

NOTE For information about using cookies with servlets, see Chapter 38.

TCP/IP Server Sockets

As mentioned earlier, Java has a different socket class that must be used for creating server applications. The **ServerSocket** class is used to create servers that listen for either local or remote client programs to connect to them on published ports. **ServerSocket**s are quite different from normal **Socket**s. When you create a **ServerSocket,** it will register itself with the system as having an interest in client connections. The constructors for **ServerSocket** reflect the port number that you want to accept connections on and, optionally, how long you want the queue for said port to be. The queue length tells the system how many client connections it can leave pending before it should simply refuse connections. The default

is 50. The constructors might throw an **IOException** under adverse conditions. Here are three of its constructors:

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

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

Datagrams

TCP/IP-style networking is appropriate for most networking needs. It provides a serialized, predictable, reliable stream of packet data. This is not without its cost, however. TCP includes many complicated algorithms for dealing with congestion control on crowded networks, as well as pessimistic expectations about packet loss. This leads to a somewhat inefficient way to transport data. Datagrams provide an alternative.

Datagrams are bundles of information passed between machines. They are somewhat like a hard throw from a well-trained but blindfolded catcher to the third baseman. Once the datagram has been released to its intended target, there is no assurance that it will arrive or even that someone will be there to catch it. Likewise, when the datagram is received, there is no assurance that it hasn't been damaged in transit or that whoever sent it is still there to receive a response.

Java implements datagrams on top of the UDP protocol by using two classes: the **DatagramPacket** object is the data container, while the **DatagramSocket** is the mechanism used to send or receive the **DatagramPacket**s. Each is examined here.

DatagramSocket

DatagramSocket defines four public constructors. They are shown here:

DatagramSocket( ) throws SocketException

DatagramSocket(int *port*) throws SocketException

DatagramSocket(int *port*, InetAddress *ipAddress*) throws SocketException

DatagramSocket(SocketAddress *address*) throws SocketException

The first creates a **DatagramSocket** bound to any unused port on the local computer. The second creates a **DatagramSocket** bound to the port specified by *port*. The third constructs a **DatagramSocket** bound to the specified port and **InetAddress**. The fourth constructs a **DatagramSocket** bound to the specified **SocketAddress**. **SocketAddress** is an abstract

class that is implemented by the concrete class **InetSocketAddress**. **InetSocketAddress** encapsulates an IP address with a port number. All can throw a **SocketException** if an error occurs while creating the socket.

DatagramSocket defines many methods. Two of the most important are **send( )** and **receive( )**, which are shown here:

void send(DatagramPacket *packet*) throws IOException

void receive(DatagramPacket *packet*) throws IOException

The **send( )** method sends a packet to the port specified by *packet*. The **receive( )** method waits for a packet to be received and returns the result.

DatagramSocket also defines the **close( )**method, which closes the socket. Beginning with JDK 7, **DatagramSocket** implements **AutoCloseable**, which means that a **DatagramSocket** can be managed by a **try**-with-resources block.

Other methods give you access to various attributes associated with a **DatagramSocket**. Here is a sampling:

InetAddress getInetAddress( )	If the socket is connected, then the address is returned. Otherwise, **null** is returned.
int getLocalPort( )	Returns the number of the local port.
int getPort( )	Returns the number of the port to which the socket is connected. It returns –1 if the socket is not connected to a port.
boolean isBound( )	Returns **true** if the socket is bound to an address. Returns **false** otherwise.
boolean isConnected( )	Returns **true** if the socket is connected to a server. Returns **false** otherwise.
void setSoTimeout(int *millis*) throws SocketException	Sets the time-out period to the number of milliseconds passed in *millis*.

DatagramPacket

DatagramPacket defines several constructors. Four are shown here:

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

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

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

InetAddress getAddress( )	Returns the address of the source (for datagrams being received) or destination (for datagrams being sent).
byte[ ] getData( )	Returns the byte array of data contained in the datagram. Mostly used to retrieve data from the datagram after it has been received.
int getLength( )	Returns the length of the valid data contained in the byte array that would be returned from the **getData( )** method. This may not equal the length of the whole byte array.
int getOffset( )	Returns the starting index of the data.
int getPort( )	Returns the port number.
void setAddress(InetAddress *ipAddress*)	Sets the address to which a packet will be sent. The address is specified by *ipAddress*.
void setData(byte[ ] *data*)	Sets the data to *data*, the offset to zero, and the length to number of bytes in *data*.
void setData(byte[ ] *data*, int *idx*, int *size*)	Sets the data to *data*, the offset to *idx*, and the length to *size*.
void setLength(int *size*)	Sets the length of the packet to *size*.
void setPort(int *port*)	Sets the port to *port*.

A Datagram Example

The following example implements a very simple networked communications client and server. Messages are typed into the window at the server and written across the network to the client side, where they are displayed.

```
// Demonstrate datagrams.
import java.net.*;

class WriteServer {
  public static int serverPort = 998;
  public static int clientPort = 999;
  public static int buffer_size = 1024;
  public static DatagramSocket ds;
  public static byte buffer[] = new byte[buffer_size];

  public static void TheServer() throws Exception {
    int pos=0;
    while (true) {
      int c = System.in.read();
      switch (c) {
        case -1:
          System.out.println("Server Quits.");
```

```
            ds.close();
            return;
          case '\r':
            break;
          case '\n':
            ds.send(new DatagramPacket(buffer,pos,
               InetAddress.getLocalHost(),clientPort));
            pos=0;
            break;
          default:
            buffer[pos++] = (byte) c;
        }
      }
    }

    public static void TheClient() throws Exception {
      while(true) {
        DatagramPacket p = new DatagramPacket(buffer, buffer.length);
        ds.receive(p);
        System.out.println(new String(p.getData(), 0, p.getLength()));
      }
    }

    public static void main(String args[]) throws Exception {
      if(args.length == 1) {
        ds = new DatagramSocket(serverPort);
        TheServer();
      } else {
        ds = new DatagramSocket(clientPort);
        TheClient();
      }
    }
}
```

This sample program is restricted by the **DatagramSocket** constructor to running between two ports on the local machine. To use the program, run

```
java WriteServer
```

in one window; this will be the client. Then run

```
java WriteServer 1
```

This will be the server. Anything that is typed in the server window will be sent to the client window after a newline is received.

NOTE The use of datagrams may not be allowed on your computer. (For example, a firewall may prevent their use.) If this is the case, the preceding example cannot be used. Also, the port numbers used in the program work on the author's system, but may have to be adjusted for your environment.

This page has been intentionally left blank

CHAPTER

23 The Applet Class

This chapter examines the **Applet** class, which provides the foundation for applets. The **Applet** class is contained in the **java.applet** package. **Applet** contains several methods that give you detailed control over the execution of your applet. In addition, **java.applet** also defines three interfaces: **AppletContext**, **AudioClip**, and **AppletStub**.

Two Types of Applets

It is important to state at the outset that there are two varieties of applets based on **Applet**. The first are those based directly on the **Applet** class described in this chapter. These applets use the Abstract Window Toolkit (AWT) to provide the graphical user interface (or use no GUI at all). This style of applet has been available since Java was first created.

The second type of applets are those based on the Swing class **JApplet**, which inherits **Applet**. Swing applets use the Swing classes to provide the GUI. Swing offers a richer and often easier-to-use user interface than does the AWT. Thus, Swing-based applets are now the most popular. However, traditional AWT-based applets are still used, especially when only a very simple user interface is required. Thus, both AWT- and Swing-based applets are valid.

This chapter describes AWT-based applets. However, because **JApplet** inherits **Applet**, all the features of **Applet** are also available in **JApplet**, and much of the information in this chapter applies to both types of applets. Therefore, even if you are interested in only Swing applets, the information in this chapter is still relevant and necessary. Understand, however, that when creating Swing-based applets, some additional constraints apply and these are described later in this book, when Swing is covered.

NOTE For information on building applets when using Swing, see Chapter 31.

Applet Basics

Chapter 13 introduced the general form of an applet and the steps necessary to compile and run one. Let's begin by reviewing this information.

AWT-based applets are subclasses of **Applet**. Applets are not stand-alone programs. Instead, they run within either a web browser or an applet viewer. The illustrations shown in this chapter were created with the standard applet viewer, called **appletviewer**, provided by the JDK.

Execution of an applet does not begin at **main( )**. Actually, few applets even have **main( )** methods. Instead, execution of an applet is started and controlled with an entirely different mechanism, which will be explained shortly. Output to your applet's window is not performed by **System.out.println( )**. Rather, in an AWT-based applet, output is handled with various AWT methods, such as **drawString( )**, which outputs a string to a specified X,Y location. Input is also handled differently than in a console application.

Before an applet can be used, a deployment strategy must be chosen. There are two basic approaches. The first is to use the Java Network Launch Protocol (JNLP). This approach offers the most flexibility, especially as it relates to rich Internet applications. For real-world applets that you create, JNLP will often be the best choice. However, a detailed discussion of JNLP is beyond the scope of this book. (See the JDK documentation for the latest details on JNLP.) Fortunately, JNLP is not required for the example applets shown here.

The second basic approach to deploying an applet is to specify the applet directly in an HTML file, without the use of JNLP. This is the original way that applets were launched when Java was created, and it is still used today—especially for simple applets. Furthermore, because of its inherent simplicity, it is the appropriate method for the applet examples described in this book. At the time of this writing, Oracle recommends the APPLET tag for this purpose. Therefore, the APPLET tag is used in this book. (Be aware that the APPLET tag is currently deprecated by the HTML specification. The alternative is the OBJECT tag. You should check the JDK documentation in this regard for the latest recommendations.) When an APPLET tag is encountered in the HTML file, the specified applet will be executed by a Java-enabled web browser.

The use of the APPLET tag offers a secondary advantage when developing applets because it enables you to easily view and test the applet. To do so, simply include a comment at the head of your Java source code file that contains the APPLET tag. This way, your code is documented with the necessary HTML statements needed by your applet, and you can test the compiled applet by starting the applet viewer with your Java source code file specified as the target. Here is an example of such a comment:

```
/*
<applet code="MyApplet" width=200 height=60>
</applet>
*/
```

This comment contains an APPLET tag that will run an applet called **MyApplet** in a window that is 200 pixels wide and 60 pixels high. Because the inclusion of an APPLET command makes testing applets easier, all of the applets shown in this book will contain the appropriate APPLET tag embedded in a comment.

NOTE As noted in Chapter 13, beginning with the release of Java 7, update 21, Java applets must be signed to prevent security warnings when run in a browser. In fact, in some cases, the applet may be prevented from running. Applets stored in the local file system, such as you would create when compiling the examples in this book, are especially sensitive to this change. You may need to adjust the security settings in the Java Control Panel to run a local applet in a browser. At the time of this writing, Oracle recommends against the use of local applets, recommending instead that applets be executed through a web server. Furthermore, unsigned local applets may be blocked from execution in the future. In general, for applets that will be distributed via the Internet, such as commercial

applications, signing is a virtual necessity. The concepts and techniques required to sign applets (and other types of Java programs) are beyond the scope of this book. However, extensive information is found on Oracle's website. Finally, as mentioned, the easiest way to try the applet examples is to use **appletviewer**.

The Applet Class

The **Applet** class defines the methods shown in Table 23-1. **Applet** provides all necessary support for applet execution, such as starting and stopping. It also provides methods that load and display images, and methods that load and play audio clips. **Applet** extends the AWT class **Panel**. In turn, **Panel** extends **Container**, which extends **Component**. These classes provide support for Java's window-based, graphical interface. Thus, **Applet** provides all of the necessary support for window-based activities. (An overview of the AWT is presented in subsequent chapters.)

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

Table 23-1 The Methods Defined by **Applet**

Method	Description
String[ ] [ ] getParameterInfo( )	Overrides of this method should return a **String** table that describes the parameters recognized by the applet. Each entry in the table must consist of three strings that contain the name of the parameter, a description of its type and/or range, and an explanation of its purpose. The default implementation returns **null**.
void init( )	Called when an applet begins execution. It is the first method called for any applet.
boolean isActive( )	Returns **true** if the applet has been started. It returns **false** if the applet has been stopped.
boolean isValidateRoot( )	Returns **true**, which indicates that an applet is a validate root.
static final AudioClip newAudioClip(URL *url*)	Returns an **AudioClip** object that encapsulates the audio clip found at the location specified by *url*. This method is similar to **getAudioClip( )** except that it is static and can be executed without the need for an **Applet** object.
void play(URL *url*)	If an audio clip is found at the location specified by *url*, the clip is played.
void play(URL *url*, String *clipName*)	If an audio clip is found at the location specified by *url* with the name specified by *clipName*, the clip is played.
void resize(Dimension *dim*)	Resizes the applet according to the dimensions specified by *dim*. **Dimension** is a class stored inside **java.awt**. It contains two integer fields: **width** and **height**.
void resize(int *width*, int *height*)	Resizes the applet according to the dimensions specified by *width* and *height*.
final void setStub(AppletStub *stubObj*)	Makes *stubObj* the stub for the applet. This method is used by the run-time system and is not usually called by your applet. A *stub* is a small piece of code that provides the linkage between your applet and the browser.
void showStatus(String *str*)	Displays *str* in the status window of the browser or applet viewer. If the browser does not support a status window, then no action takes place.
void start( )	Called by the browser when an applet should start (or resume) execution. It is automatically called after **init( )** when an applet first begins.
void stop( )	Called by the browser to suspend execution of the applet. Once stopped, an applet is restarted when the browser calls **start( )**.

Table 23-1 The Methods Defined by **Applet** *(continued)*

Applet Architecture

As a general rule, an applet is a GUI-based program. As such, its architecture is different from the console-based programs shown in the first part of this book. If you are already familiar with GUI programming, you will be right at home writing applets. If not, then there are a few key concepts you must understand.

First, applets are event driven. Although we won't examine event handling until the following chapter, it is important to understand in a general way how the event-driven architecture impacts the design of an applet. An applet resembles a set of interrupt service routines. Here is how the process works. An applet waits until an event occurs. The run-time system notifies the applet about an event by calling an event handler that has been provided by the applet. Once this happens, the applet must take appropriate action and then quickly return. This is a crucial point. For the most part, your applet should not enter a "mode" of operation in which it maintains control for an extended period. Instead, it must perform specific actions in response to events and then return control to the run-time system. In those situations in which your applet needs to perform a repetitive task on its own (for example, displaying a scrolling message across its window), you must start an additional thread of execution. (You will see an example later in this chapter.)

Second, the user initiates interaction with an applet—not the other way around. As you know, in a console-based program, when the program needs input, it will prompt the user and then call some input method, such as **readLine( )**. This is not the way it works in an applet. Instead, the user interacts with the applet as he or she wants, when he or she wants. These interactions are sent to the applet as events to which the applet must respond. For example, when the user clicks the mouse inside the applet's window, a mouse-clicked event is generated. If the user presses a key while the applet's window has input focus, a keypress event is generated. As you will see in later chapters, applets can contain various controls, such as push buttons and check boxes. When the user interacts with one of these controls, an event is generated.

While the architecture of an applet is not as easy to understand as that of a console-based program, Java makes it as simple as possible. If you have written programs for Windows (or other GUI-based operating systems), you know how intimidating that environment can be. Fortunately, Java provides a much cleaner approach that is more quickly mastered.

An Applet Skeleton

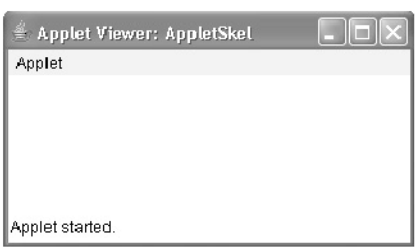
All but the most trivial applets override a set of methods that provides the basic mechanism by which the browser or applet viewer interfaces to the applet and controls its execution. Four of these methods, **init( )**, **start( )**, **stop( )**, and **destroy( )**, apply to all applets and are defined by **Applet**. Default implementations for all of these methods are provided. Applets do not need to override those methods they do not use. However, only very simple applets will not need to define all of them.

AWT-based applets (such as those discussed in this chapter) will also often override the **paint( )** method, which is defined by the AWT **Component** class. This method is called when

the applet's output must be redisplayed. (Swing-based applets use a different mechanism to accomplish this task.) These five methods can be assembled into the skeleton shown here:

```
// An Applet skeleton.
import java.awt.*;
import java.applet.*;
/*
<applet code="AppletSkel" width=300 height=100>
</applet>
*/

public class AppletSkel extends Applet {
  // Called first.
  public void init() {
    // initialization
  }

  /* Called second, after init(). Also called whenever
     the applet is restarted. */
  public void start() {
    // start or resume execution
  }

  // Called when the applet is stopped.
  public void stop() {
    // suspends execution
  }

  /* Called when applet is terminated. This is the last
     method executed. */
  public void destroy() {
    // perform shutdown activities
  }

  // Called when an applet's window must be restored.
  public void paint(Graphics g) {
    // redisplay contents of window
  }
}
```

Although this skeleton does not do anything, it can be compiled and run. When run, it generates the following empty window when viewed with **appletviewer**. Of course, in this and all subsequent examples, the precise look of the **appletviewer** frame may differ based on your execution environment. To help illustrate this fact, a variety of environments were used to generate the screen captures shown throughout this book.



Applet Initialization and Termination

It is important to understand the order in which the various methods shown in the skeleton are called. When an applet begins, the following methods are called, in this sequence:

1. **init( )**
2. **start( )**
3. **paint( )**

When an applet is terminated, the following sequence of method calls takes place:

1. **stop( )**
2. **destroy( )**

Let's look more closely at these methods.

init( )

The **init( )** method is the first method to be called. This is where you should initialize variables. This method is called only once during the run time of your applet.

start( )

The **start( )** method is called after **init( )**. It is also called to restart an applet after it has been stopped. Whereas **init( )** is called once—the first time an applet is loaded—**start( )** is called each time an applet's HTML document is displayed onscreen. So, if a user leaves a web page and comes back, the applet resumes execution at **start( )**.

paint( )

The **paint( )** method is called each time an AWT-based applet's output must be redrawn. This situation can occur for several reasons. For example, the window in which the applet is running may be overwritten by another window and then uncovered. Or the applet window may be minimized and then restored. **paint( )** is also called when the applet begins execution. Whatever the cause, whenever the applet must redraw its output, **paint( )** is called. The **paint( )** method has one parameter of type **Graphics**. This parameter will contain the graphics context, which describes the graphics environment in which the applet is running. This context is used whenever output to the applet is required.

stop( )

The **stop( )** method is called when a web browser leaves the HTML document containing the applet—when it goes to another page, for example. When **stop( )** is called, the applet is probably running. You should use **stop( )** to suspend threads that don't need to run when the applet is not visible. You can restart them when **start( )** is called if the user returns to the page.

destroy( )

The **destroy( )** method is called when the environment determines that your applet needs to be removed completely from memory. At this point, you should free up any resources the applet may be using. The **stop( )** method is always called before **destroy( )**.

Overriding update( )

In some situations, an AWT-based applet may need to override another method defined by the AWT, called **update( )**. This method is called when your applet has requested that a portion of its window be redrawn. The default version of **update( )** simply calls **paint( )**. However, you can override the **update( )** method so that it performs more subtle repainting. In general, overriding **update( )** is a specialized technique that is not applicable to all applets, and the examples in this chapter do not override **update( )**.

Simple Applet Display Methods

As we've mentioned, applets are displayed in a window, and AWT-based applets use the AWT to perform input and output. Although we will examine the methods, procedures, and techniques related to the AWT in subsequent chapters, a few are described here, because we will use them to write sample applets. (Remember, Swing-based applets are described later in this book.)

As described in Chapter 13, to output a string to an applet, use **drawString( )**, which is a member of the **Graphics** class. Typically, it is called from within either **update( )** or **paint( )**. It has the following general form:

void drawString(String *message*, int *x*, int *y*)

Here, *message* is the string to be output beginning at *x,y*. In a Java window, the upper-left corner is location 0,0. The **drawString( )** method will not recognize newline characters. If you want to start a line of text on another line, you must do so manually, specifying the precise X,Y location where you want the line to begin. (As you will see in later chapters, there are techniques that make this process easy.)

To set the background color of an applet's window, use **setBackground( )**. To set the foreground color (the color in which text is shown, for example), use **setForeground( )**. These methods are defined by **Component**, and they have the following general forms:

void setBackground(Color *newColor*)
void setForeground(Color *newColor*)

Here, *newColor* specifies the new color. The class **Color** defines the constants shown here that can be used to specify colors:

Color.black	Color.magenta
Color.blue	Color.orange
Color.cyan	Color.pink
Color.darkGray	Color.red
Color.gray	Color.white
Color.green	Color.yellow
Color.lightGray	

Uppercase versions of the constants are also defined.

The following example sets the background color to green and the text color to red:

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

A good place to set the foreground and background colors is in the **init( )** method. Of course, you can change these colors as often as necessary during the execution of your applet.

You can obtain the current settings for the background and foreground colors by calling **getBackground( )** and **getForeground( )**, respectively. They are also defined by **Component** and are shown here:

Color getBackground( )
Color getForeground( )

Here is a very simple applet that sets the background color to cyan, the foreground color to red, and displays a message that illustrates the order in which the **init( )**, **start( )**, and **paint( )** methods are called when an applet starts up:

```
/* A simple applet that sets the foreground and
   background colors and outputs a string. */
import java.awt.*;
import java.applet.*;
/*
<applet code="Sample" width=300 height=50>
</applet>
*/

public class Sample extends Applet{
  String msg;

  // set the foreground and background colors.
  public void init() {
    setBackground(Color.cyan);
    setForeground(Color.red);
    msg = "Inside init( ) --";
  }

  // Initialize the string to be displayed.
  public void start() {
    msg += " Inside start( ) --";
  }

  // Display msg in applet window.
  public void paint(Graphics g) {
    msg += " Inside paint( ).";
    g.drawString(msg, 10, 30);
  }
}
```

Sample output is shown here:

The methods **stop( )** and **destroy( )** are not overridden, because they are not needed by this simple applet.

Requesting Repainting

As a general rule, an applet writes to its window only when its **paint( )** method is called by the AWT. This raises an interesting question: How can the applet itself cause its window to be updated when its information changes? For example, if an applet is displaying a moving banner, what mechanism does the applet use to update the window each time this banner scrolls? Remember, one of the fundamental architectural constraints imposed on an applet is that it must quickly return control to the run-time system. It cannot create a loop inside **paint( )** that repeatedly scrolls the banner, for example. This would prevent control from passing back to the AWT. Given this constraint, it may seem that output to your applet's window will be difficult at best. Fortunately, this is not the case. Whenever your applet needs to update the information displayed in its window, it simply calls **repaint( )**.

The **repaint( )** method is defined by the AWT. It causes the AWT run-time system to execute a call to your applet's **update( )** method, which, in its default implementation, calls **paint( )**. Thus, for another part of your applet to output to its window, simply store the output and then call **repaint( )**. The AWT will then execute a call to **paint( )**, which can display the stored information. For example, if part of your applet needs to output a string, it can store this string in a **String** variable and then call **repaint( )**. Inside **paint( )**, you will output the string using **drawString( )**.

The **repaint( )** method has four forms. Let's look at each one, in turn. The simplest version of **repaint( )** is shown here:

```
void repaint( )
```

This version causes the entire window to be repainted. The following version specifies a region that will be repainted:

```
void repaint(int left, int top, int width, int height)
```

Here, the coordinates of the upper-left corner of the region are specified by *left* and *top*, and the width and height of the region are passed in *width* and *height*. These dimensions are specified in pixels. You save time by specifying a region to repaint. Window updates are costly in terms of time. If you need to update only a small portion of the window, it is more efficient to repaint only that region.

Calling **repaint( )** is essentially a request that your applet be repainted sometime soon. However, if your system is slow or busy, **update( )** might not be called immediately. Multiple requests for repainting that occur within a short time can be collapsed by the AWT in a manner such that **update( )** is only called sporadically. This can be a problem in many situations, including animation, in which a consistent update time is necessary. One solution to this problem is to use the following forms of **repaint( )**:

void repaint(long *maxDelay*)
void repaint(long *maxDelay*, int *x*, int *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.

NOTE It is possible for a method other than **paint( )** or **update( )** to output to an applet's window. To do so, it must obtain a graphics context by calling **getGraphics( )** (defined by **Component**) and then use this context to output to the window. However, for most applications, it is better and easier to route window output through **paint( )** and to call **repaint( )** when the contents of the window change.

A Simple Banner Applet

To demonstrate **repaint( )**, a simple banner applet is developed. This applet scrolls a message, from right to left, across the applet's window. Since the scrolling of the message is a repetitive task, it is performed by a separate thread, created by the applet when it is initialized. The banner applet is shown here:

```
/* A simple banner applet.

   This applet creates a thread that scrolls
   the message contained in msg right to left
   across the applet's window.
*/
import java.awt.*;
import java.applet.*;
/*
<applet code="SimpleBanner" width=300 height=50>
</applet>
*/

public class SimpleBanner extends Applet implements Runnable {
  String msg = " A Simple Moving Banner.";
  Thread t = null;
  int state;
  volatile boolean stopFlag;

  // Set colors and initialize thread.
  public void init() {
    setBackground(Color.cyan);
    setForeground(Color.red);
  }
```

```
  // Start thread
  public void start() {
    t = new Thread(this);
    stopFlag = false;
    t.start();
  }

  // Entry point for the thread that runs the banner.
  public void run() {

    // Redisplay banner
    for( ; ; ) {
      try {
        repaint();
        Thread.sleep(250);
        if(stopFlag)
          break;
      } catch(InterruptedException e) {}
    }
  }

  // Pause the banner.
  public void stop() {
    stopFlag = true;
    t = null;
  }

  // Display the banner.
  public void paint(Graphics g) {
    char ch;

    ch = msg.charAt(0);
    msg = msg.substring(1, msg.length());
    msg += ch;

    g.drawString(msg, 50, 30);
  }
}
```

Following is sample output:

Let's take a close look at how this applet operates. First, notice that **SimpleBanner** extends **Applet**, as expected, but it also implements **Runnable**. This is necessary, since the

applet will be creating a second thread of execution that will be used to scroll the banner. Inside **init( )**, the foreground and background colors of the applet are set.

After initialization, the run-time system calls **start( )** to start the applet running. Inside **start( )**, a new thread of execution is created and assigned to the **Thread** variable **t**. Then, the **boolean** variable **stopFlag**, which controls the execution of the applet, is set to **false**. Next, the thread is started by a call to **t.start( )**. Remember that **t.start( )** calls a method defined by **Thread**, which causes **run( )** to begin executing. It does not cause a call to the version of **start( )** defined by **Applet**. These are two separate methods.

Inside **run( )**, a call to **repaint( )** is made. This eventually causes the **paint( )** method to be called, and the rotated contents of **msg** are displayed. Between each iteration, **run( )** sleeps for a quarter of a second. The net effect is that the contents of **msg** are scrolled right to left in a constantly moving display. The **stopFlag** variable is checked on each iteration. When it is **true**, the **run( )** method terminates.

If a browser is displaying the applet when a new page is viewed, the **stop( )** method is called, which sets **stopFlag** to **true**, causing **run( )** to terminate. This is the mechanism used to stop the thread when its page is no longer in view. When the applet is brought back into view, **start( )** is once again called, which starts a new thread to execute the banner.

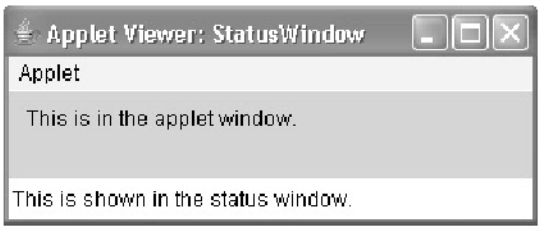
Using the Status Window

In addition to displaying information in its window, an applet can also output a message to the status window of the browser or applet viewer on which it is running. To do so, call **showStatus( )** with the string that you want displayed. The status window is a good place to give the user feedback about what is occurring in the applet, suggest options, or possibly report some types of errors. The status window also makes an excellent debugging aid, because it gives you an easy way to output information about your applet.

The following applet demonstrates **showStatus( )**:

```
// Using the Status Window.
import java.awt.*;
import java.applet.*;
/*

<applet code="StatusWindow" width=300 height=50>
</applet>
*/

public class StatusWindow extends Applet {
  public void init() {
    setBackground(Color.cyan);
  }

  // Display msg in applet window.
  public void paint(Graphics g) {
    g.drawString("This is in the applet window.", 10, 20);
    showStatus("This is shown in the status window.");
  }
}
```

Sample output from this program is shown here:



The HTML APPLET Tag

As mentioned earlier, at the time of this writing, Oracle recommends that the APPLET tag be used to manually start an applet when JNLP is not used. An applet viewer will execute each APPLET tag that it finds in a separate window, while web browsers will allow many applets on a single page. So far, we have been using only a simplified form of the APPLET tag. Now it is time to take a closer look at it.

The syntax for a fuller form of the APPLET tag is shown here. Bracketed items are optional.

```
< APPLET
  [CODEBASE = codebaseURL]
  CODE = appletFile
  [ALT = alternateText]
  [NAME = appletInstanceName]
  WIDTH = pixels HEIGHT = pixels
  [ALIGN = alignment ]
  [VSPACE = pixels] [HSPACE = pixels]
>
[< PARAM NAME = AttributeName VALUE = AttributeValue>]
[< PARAM NAME = AttributeName2 VALUE = AttributeValue>]
. . .
[HTML Displayed in the absence of Java]
</APPLET>
```

Let's take a look at each part now.

CODEBASE CODEBASE is an optional attribute that specifies the base URL of the applet code, which is the directory that will be searched for the applet's executable class file (specified by the CODE tag). The HTML document's URL directory is used as the CODEBASE if this attribute is not specified.

CODE CODE is a required attribute that gives the name of the file containing your applet's compiled **.class** file. This file is relative to the code base URL of the applet, which is the directory that the HTML file was in or the directory indicated by CODEBASE if set.

ALT The ALT tag is an optional attribute used to specify a short text message that should be displayed if the browser recognizes the APPLET tag but can't currently run Java applets. This is distinct from the alternate HTML you provide for browsers that don't support applets.

NAME NAME is an optional attribute used to specify a name for the applet instance. Applets must be named in order for other applets on the same page to find them by name and communicate with them. To obtain an applet by name, use **getApplet( )**, which is defined by the **AppletContext** interface.

WIDTH and HEIGHT WIDTH and HEIGHT are required attributes that give the size (in pixels) of the applet display area.

ALIGN ALIGN is an optional attribute that specifies the alignment of the applet. This attribute is treated the same as the HTML IMG tag with these possible values: LEFT, RIGHT, TOP, BOTTOM, MIDDLE, BASELINE, TEXTTOP, ABSMIDDLE, and ABSBOTTOM.

VSPACE and HSPACE These attributes are optional. VSPACE specifies the space, in pixels, above and below the applet. HSPACE specifies the space, in pixels, on each side of the applet. They're treated the same as the IMG tag's VSPACE and HSPACE attributes.

PARAM NAME and VALUE The PARAM tag allows you to specify applet-specific arguments. Applets access their attributes with the **getParameter( )** method.

Other valid APPLET attributes include ARCHIVE, which lets you specify one or more archive files, and OBJECT, which specifies a saved version of the applet. In general, an APPLET tag should include only a CODE or an OBJECT attribute, but not both.

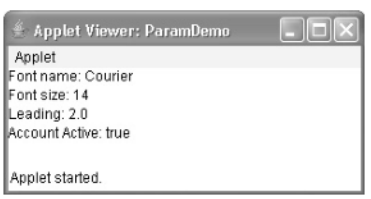
Passing Parameters to Applets

As just discussed, the APPLET tag allows you to pass parameters to your applet. To retrieve a parameter, use the **getParameter( )** method. It returns the value of the specified parameter in the form of a **String** object. Thus, for numeric and **boolean** values, you will need to convert their string representations into their internal formats. Here is an example that demonstrates passing parameters:

```
// Use Parameters
import java.awt.*;
import java.applet.*;
/*
<applet code="ParamDemo" width=300 height=80>
<param name=fontName value=Courier>
<param name=fontSize value=14>
<param name=leading value=2>
<param name=accountEnabled value=true>
</applet>
*/

public class ParamDemo extends Applet {
  String fontName;
```

```
  int fontSize;
  float leading;
  boolean active;

  // Initialize the string to be displayed.
  public void start() {
    String param;

    fontName = getParameter("fontName");
    if(fontName == null)
      fontName = "Not Found";

    param = getParameter("fontSize");
    try {
      if(param != null)
        fontSize = Integer.parseInt(param);
      else
        fontSize = 0;
    } catch(NumberFormatException e) {
      fontSize = -1;
    }

    param = getParameter("leading");
    try {
      if(param != null)
        leading = Float.valueOf(param).floatValue();
      else
        leading = 0;
    } catch(NumberFormatException e) {
      leading = -1;
    }

    param = getParameter("accountEnabled");
    if(param != null)
      active = Boolean.valueOf(param).booleanValue();
  }

  // Display parameters.
  public void paint(Graphics g) {
    g.drawString("Font name: " + fontName, 0, 10);
    g.drawString("Font size: " + fontSize, 0, 26);
    g.drawString("Leading: " + leading, 0, 42);
    g.drawString("Account Active: " + active, 0, 58);
  }
}
```

Sample output from this program is shown here:



As the program shows, you should test the return values from **getParameter( )**. If a parameter isn't available, **getParameter( )** will return **null**. Also, conversions to numeric types must be attempted in a **try** statement that catches **NumberFormatException**. Uncaught exceptions should never occur within an applet.

Improving the Banner Applet

It is possible to use a parameter to enhance the banner applet shown earlier. In the previous version, the message being scrolled was hard-coded into the applet. However, passing the message as a parameter allows the banner applet to display a different message each time it is executed. This improved version is shown here. Notice that the APPLET tag at the top of the file now specifies a parameter called **message** that is linked to a quoted string.

```
// A parameterized banner
import java.awt.*;
import java.applet.*;
/*
<applet code="ParamBanner" width=300 height=50>
<param name=message value="Java makes the Web move!">
</applet>
*/

public class ParamBanner extends Applet implements Runnable {
  String msg;
  Thread t = null;
  int state;
  volatile boolean stopFlag;

  // Set colors and initialize thread.
  public void init() {
    setBackground(Color.cyan);
    setForeground(Color.red);
  }

  // Start thread
  public void start() {
    msg = getParameter("message");
    if(msg == null) msg = "Message not found.";
    msg = " " + msg;
    t = new Thread(this);
    stopFlag = false;
    t.start();
  }

  // Entry point for the thread that runs the banner.
  public void run() {

    // Redisplay banner
    for( ; ; ) {
      try {
        repaint();
        Thread.sleep(250);
        if(stopFlag)
          break;
      } catch(InterruptedException e) {}
    }
  }
```

```
  // Pause the banner.
  public void stop() {
    stopFlag = true;
    t = null;
  }

  // Display the banner.
  public void paint(Graphics g) {
    char ch;

    ch = msg.charAt(0);
    msg = msg.substring(1, msg.length());
    msg += ch;

    g.drawString(msg, 50, 30);
  }
}
```

getDocumentBase( ) and getCodeBase( )

Often, you will create applets that will need to explicitly load media and text. Java will allow the applet to load data from the directory holding the HTML file that started the applet (the *document base*) and the directory from which the applet's class file was loaded (the *code base*). These directories are returned as **URL** objects (described in Chapter 22) by **getDocumentBase( )** and **getCodeBase( )**. They can be concatenated with a string that names the file you want to load. To actually load another file, you will use the **showDocument( )** method defined by the **AppletContext** interface, discussed in the next section.

The following applet illustrates these methods:

```
// Display code and document bases.
import java.awt.*;
import java.applet.*;
import java.net.*;
/*
<applet code="Bases" width=300 height=50>
</applet>
*/

public class Bases extends Applet {
  // Display code and document bases.
  public void paint(Graphics g) {
    String msg;

    URL url = getCodeBase(); // get code base
    msg = "Code base: " + url.toString();
    g.drawString(msg, 10, 20);

    url = getDocumentBase(); // get document base
    msg = "Document base: " + url.toString();
    g.drawString(msg, 10, 40);
  }
}
```

Sample output from this program is shown here:

AppletContext and showDocument( )

One application of Java is to use active images and animation to provide a graphical means of navigating the Web that is more interesting than simple text-based links. To allow your applet to transfer control to another URL, you must use the **showDocument( )** method defined by the **AppletContext** interface. **AppletContext** is an interface that lets you get information from the applet's execution environment. The methods defined by **AppletContext** are shown in Table 23-2. The context of the currently executing applet is obtained by a call to the **getAppletContext( )** method defined by **Applet**.

Method	Description
Applet getApplet(String *appletName*)	Returns the applet specified by *appletName* if it is within the current applet context. Otherwise, **null** is returned.
Enumeration<Applet> getApplets( )	Returns an enumeration that contains all of the applets within the current applet context.
AudioClip getAudioClip(URL *url*)	Returns an **AudioClip** object that encapsulates the audio clip found at the location specified by *url*.
Image getImage(URL *url*)	Returns an **Image** object that encapsulates the image found at the location specified by *url*.
InputStream getStream(String *key*)	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*.
Iterator<String> getStreamKeys( )	Returns an iterator for the keys associated with the invoking object. The keys are linked to streams. See **getStream( )** and **setStream( )**.
void setStream(String *key*, InputStream *strm*) throws IOException	Links the stream specified by *strm* to the key passed in *key*. The *key* is deleted from the invoking object if *strm* is **null**.
void showDocument(URL *url*)	Brings the document at the **URL** specified by *url* into view. This method may not be supported by applet viewers.

Table 23-2 The Methods Defined by the **AppletContext** Interface