APPENDIX A
Content and Protocol Handlers

Content and protocol handlers represent one of the most interesting ideas from the original Java vision. Unfortunately, as far as we can tell, no one has taken up the challenge of using this intriguing facility. We considered dropping them from the book entirely, but that decision just felt bad. Instead, we banished the discussion of how to write content and protocol handlers to an appendix. If you let us know that this material is important to you, we'll keep it in the next edition. If you feel "yes, this is interesting, but why do I care?" we'll drop them from the book. (You can send comments via the book's web page at *http://www.oreilly.com/catalog/learnjava2*.)

This appendix picks up where we left our discussion of content and protocol handlers in Chapter 13. We'll show you how to write your own handlers, which can be used in any Java application, including the HotJava web browser. In this section, we'll write a content handler that reads Unix tar files and a protocol handler that implements a pluggable encryption scheme. You should be able to drop both into your class path and start using them in the HotJava web browser right away.

Writing a Content Handler

The URL class's `getContent( )` method invokes a content handler whenever it's called to retrieve an object at some URL. The content handler must read the flat stream of data produced by the `URL`'s protocol handler (the data read from the remote source), and construct a well-defined Java object from it. By "flat," we mean that the data stream the content handler receives has no artifacts left from retrieving the data and processing the protocol. It's the protocol handler's job to fetch and decode the data before passing it along. The protocol handler's output is your data, pure and simple.

The roles of content and protocol handlers do not overlap. The content handler doesn't care how the data arrives or what form it takes. It's concerned only with what kind of object it's supposed to create. For example, if a particular protocol involves sending an object over the network in a compressed format, the protocol handler should do whatever is necessary to unpack it before passing the data on to the content handler. The same content handler can then be used again with a completely different protocol handler to construct the *same* type of object received via a *different* transport mechanism.

Let's look at an example. The following lines construct a URL that points to a GIF file on an FTP archive and attempt to retrieve its contents:

```
try {
  URL url =
      new URL ("ftp://ftp.wustl.edu/graphics/gif/a/apple.gif");
  ImageProducer imgsrc = (ImageProducer)url.getContent( );
  ...
```

When we construct the `URL` object, Java looks at the first part of the URL string (everything prior to the colon) to determine the protocol and locate a protocol handler. In this case, it locates the FTP protocol handler, which is used to open a connection to the host and transfer data for the specified file.

After making the connection, the `URL` object asks the protocol handler to identify the resource's MIME type. The handler can try to resolve the MIME type through a variety of means, but in this case, it might just look at the filename extension (*.gif*) and determine that the MIME type of the data is `image/gif`. Here, `image/gif` is a string that denotes that the content falls into the category of images and is, more specifically, a GIF image. The protocol handler then looks for the content handler responsible for the `image/gif` type and uses it to construct the right kind of object from the data. The content handler returns an `ImageProducer` object, which `getContent( )` returns to us as an `Object`. As we've seen before, we cast this `Object` back to its real type so we can work with it.

In an upcoming section, we'll build a simple content handler. To keep things simple, our example produces text as output; the `URL`'s `get-Content( )` method returns this as a `String` object.

Locating Content Handlers

When Java searches for a class, it translates package names into filesystem pathnames. (The classes may also be in a JAR file in the class path, but we refer to them as files and directories anyway.) This applies to locating content-handler classes as well as other kinds of classes. For example, a class in a package named `foo.bar.handlers` would live in a directory with *foo/bar/handlers/* as part of its pathname. To allow Java to find handler classes for arbitrary new MIME types, content handlers are organized into packages corresponding to the basic MIME type categories. The handler classes themselves are named after the specific MIME type, which allows Java to map MIME types directly to class names. The only remaining information Java needs is a list of packages in which the handlers might reside. To supply this information, you should use the system properties `java.content.handler.pkgs` and java.protocol.handler.pkgs. In these properties, you can use a vertical bar (|) to separate different packages in a list.

We'll put our content handlers in the learningjava.contenthandlers package. According to the scheme for naming content handlers, a handler for the image/gif MIME type is called gif and placed in a package that is called learningjava.contenthandlers.image. The fully qualified name of the class would then be learningjava.contenthandlers.image.gif, and it would be located in the file *learningjava/contenthandlers/image/gif.class*, somewhere in the local class path, or, perhaps someday, on a server. Likewise, a content handler for the video/mpeg MIME type would be called mpeg, and an *mpeg.class* file would be located in a *learningjava/contenthandlers/video/* directory somewhere in the class path.

Many MIME type names include a dash (-), which is illegal in a class name. You should convert dashes and other illegal characters into underscores (_) when building Java class and package names. Also note that there are no capital letters in the class names. This violates the coding convention used in most Java source files, in which class names start with capital letters. However, capitalization is not significant in MIME type names, so it is simpler to name the handler classes accordingly.

The application/x-tar Handler

In this section, we'll build a simple content handler that reads and interprets tar (tape archive) files. tar is an archival format widely used in the Unix-world to hold collections of files, along with their basic type and attribute information.* A tar file is similar to a JAR file, except that it's not compressed. Files in the archive are stored sequentially, in flat text or binary with no special encoding. In practice, tar files are usually compressed for storage using an application like Unix *compress* or GNU *gzip* and then named with a filename extension like *.tar.gz* or *.tgz*.

Most web browsers, upon retrieving a tar file, prompt the user with a File Save dialog. The assumption is that if you are retrieving an archive, you probably want to save it for later unpacking and use. We would like to implement a *tar* content handler that allows an application to read the contents of the archive and give us a listing of the files that it contains. In itself, this would not be the most useful thing in the world, because we would be left with the dilemma of how to get at the archive's contents. However, a more complete implementation of our content handler, used in conjunction with an application like a web browser, could generate HTML output or pop up a dialog that lets us select and save individual files within the archive.

* There are several slightly different versions of the tar format. This content handler understands the most widely used variant.

Some code that fetches a tar file and lists its contents might look like this:

```
try {
    URL listing =
        new URL("http://somewhere.an.edu/lynx/lynx2html.tar");
    String s = (String)listing.getContents( );
    System.out.println( s );
     ...
```

Our handler produces a listing similar to the Unix *tar* application's output:

```
Tape Archive Listing:

0     Tue Sep 28 18:12:47 CDT 1993 lynx2html/
14773 Tue Sep 28 18:01:55 CDT 1993 lynx2html/lynx2html.c
470   Tue Sep 28 18:13:24 CDT 1993 lynx2html/Makefile
172   Thu Apr 01 15:05:43 CST 1993 lynx2html/lynxgate
3656  Wed Mar 03 15:40:20 CST 1993 lynx2html/install.csh
490   Thu Apr 01 14:55:04 CST 1993 lynx2html/new_globals.c
...
```

Our handler will dissect the file to read the contents and generate the listing. The `URL`'s `getContent( )` method will return that information to an application as a `String` object.

First we must decide what to call our content handler and where to put it. The MIME-type hierarchy classifies the tar format as an *application type extension*. Its proper `MIME` type is then `application/x-tar`. Therefore, our handler belongs in the `learningjava.contenthandlers.application` package and goes into the class file *learningjava/contenthandlers/application/x_tar.class*. Note that the name of our class is `x_tar`, rather than `x-tar`; you'll remember the dash is illegal in a class name so, by convention, we convert it to an underscore.

Here's the code for the content handler; compile it and put it in *learningjava/contenthandlers/application/*, somewhere in your class path:

```
//file: x_tar.java
package learningjava.contenthandlers.application;

import java.net.*;
import java.io.*;
import java.util.Date;

public class x_tar extends ContentHandler {
  static int
    RECORDLEN = 512,
    NAMEOFF = 0, NAMELEN = 100,
    SIZEOFF = 124, SIZELEN = 12,
    MTIMEOFF = 136, MTIMELEN = 12;

  public Object getContent(URLConnection uc) throws IOException {
    InputStream is = uc.getInputStream( );
    StringBuffer output =
        new StringBuffer( "Tape Archive Listing:\n\n" );
    byte [] header = new byte[RECORDLEN];
    int count = 0;

    while ( (is.read(header) == RECORDLEN)
             && (header[NAMEOFF] != 0) ) {
      String name =
           new String(header, NAMEOFF, NAMELEN, "8859_1"). trim( );
      String s =
           new String(header, SIZEOFF, SIZELEN, "8859_1").trim( );
      int size = Integer.parseInt(s, 8);
      s = new String(header, MTIMEOFF, MTIMELEN, "8859_1").trim( );
      long l = Integer.parseInt(s, 8);
      Date mtime = new Date( l*1000 );

      output.append( size + " " + mtime + " " + name + "\n" );

      count += is.skip( size ) + RECORDLEN;
      if ( count % RECORDLEN != 0 )
        count += is.skip ( RECORDLEN - count % RECORDLEN);
    }

    if ( count == 0 )
      output.append("Not a valid TAR file\n");

    return( output.toString( ) );
  }
}
```

The ContentHandler class

Our x_tar handler is a subclass of the abstract class java.net.ContentHandler. Its job is to implement one method: getContent( ), which takes as an argument a special "protocol connection" object and returns a constructed Java Object. The getContent( ) method of the URL class ultimately uses this getContent( ) method when we ask for the contents of the URL. The code looks formidable, but most of it's involved with processing the details of the tar format. If we remove these details, there isn't much left:

```
    public class x_tar extends ContentHandler {

      public Object getContent( URLConnection uc ) throws IOException {
        // get input stream
        InputStream is = uc.getInputStream( );

        // read stream and construct object
        // ...

        // return the constructed object
        return( output.toString( ) );
      }
    }
```

That's really all there is to a content handler; it's relatively simple.

The URLConnection

The java.net.URLConnection object that getContent( ) receives represents the protocol handler's connection to the remote resource. It provides a number of methods for examining information about the URL resource, such as header and type fields, and for determining the kinds of operations the protocol supports. However, its most important method is getInputStream( ), which returns an InputStream from the protocol handler. Reading this InputStream gives you the raw data for the object the URL addresses. In our case, reading the InputStream feeds x_tar the bytes of the tar file it's to process.

Constructing the object

The majority of our getContent( ) method is devoted to interpreting the stream of bytes of the tar file and building our output object: the String that lists the contents of the tar file. Again, this means that this example involves the particulars of reading tar files, so you shouldn't fret too much about the details.

After requesting an InputStream from the URLConnection, x_tar loops, gathering information about each file. Each archived item is preceded by a header that contains attribute and length fields. x_tar interprets each header and then skips over the remaining portion of the item. To parse the header, we use the String constructor to read a fixed number of characters from the byte array header[]. To convert these bytes into a Java String properly, we specify the character encoding used by web servers: 8859_1, which (for the most part) is equivalent to ASCII. Once we have a file's name, size, and time stamp, we accumulate the results (the file listings) in a StringBuffer—one line per file. When the listing is complete, getContent( ) returns the StringBuffer as a String object.

The main while loop continues as long as it's able to read another header record, and as long as the record's "name" field isn't full of ASCII null values. (The tar file format calls for the end of the archive to be padded with an empty header record, although most tar implementations don't seem to do this.) The while loop retrieves the name, size, and modification times as character strings from fields in the header. The most common tar format stores its numeric values in octal, as fixed-length ASCII strings. We extract the strings and use Integer.parseInt( ) to parse them.

After reading and parsing the header, x_tar skips over the data portion of the file and updates the variable count, which keeps track of the offset into the archive. The two lines following the initial skip account for tar's "blocking" of the data records. In other words, if the data portion of a file doesn't fit precisely into an integral number of blocks of RECORDLEN bytes, tar adds padding to make it fit.

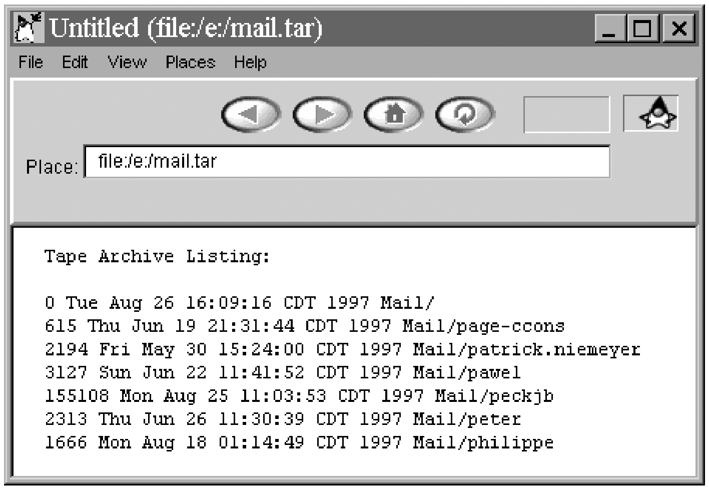
As we said, the details of parsing tar files are not really our main concern here. But x_tar does illustrate a few tricks of data manipulation in Java.

It may surprise you that we didn't have to provide a constructor; our content handler relies on its default constructor. We don't need to provide a constructor because there isn't anything for it to do. Java doesn't pass the class any argument information when it creates an instance of it. You might suspect that the `URLConnection` object would be a natural thing to provide at that point. However, when you are calling the constructor of a class that is loaded at runtime, you can't easily pass it any arguments.

Using our new handler

When we began this discussion of content handlers, we showed a brief example of how our `x_tar` content handler would work for us. You can try that code snippet now with your favorite tar file by setting the `java.content.handler.pkgs` system property to `learningjava.contenthandlers` and making sure that package is in your class path.

To make things more exciting, try setting the property in your HotJava properties file. (The HotJava properties file usually resides in a *.hotjava* directory in your home directory or in the HotJava installation directory on a Windows machine.) Make sure the class path is set before you start HotJava. Once HotJava is running, go to the *Preferences* menu, and select *Viewer Applications*. Find the type *TAR archive*, and set its *Action* to *View in HotJava*. This tells HotJava to try to use a content handler to display the data in the browser. Now, drive HotJava to a URL that contains a tar file. The result should look something like that shown in Figure A-1.



file:/e:/mail.tar

```
Tape Archive Listing:

0 Tue Aug 26 16:09:16 CDT 1997 Mail/
615 Thu Jun 19 21:31:44 CDT 1997 Mail/page-ccons
2194 Fri May 30 15:24:00 CDT 1997 Mail/patrick.niemeyer
3127 Sun Jun 22 11:41:52 CDT 1997 Mail/pawel
155108 Mon Aug 25 11:03:53 CDT 1997 Mail/peckjb
2313 Thu Jun 26 11:30:39 CDT 1997 Mail/peter
1666 Mon Aug 18 01:14:49 CDT 1997 Mail/philippe
```

Figure A-1. Using a content handler to display data in a browser

We've just extended our copy of HotJava to understand tar files! In the next section, we'll turn the tables and look at protocol handlers. There we'll be building `URLConnection` objects; someone else will have the pleasure of reconstituting the data.

Writing a Protocol Handler

A `URL` object uses a protocol handler to establish a connection with a server and perform whatever protocol is necessary to retrieve data. For example, an HTTP protocol handler knows how to talk to an HTTP server and retrieve a document; an FTP protocol handler knows how to talk to an FTP server and retrieve a file. All types of URLs use protocol handlers to access their objects. Even the lowly "file" type URLs use a special "file" protocol handler that retrieves files from the local filesystem. The data a protocol handler retrieves is then fed to an appropriate content handler for interpretation.

While we refer to a protocol handler as a single entity, it really has two parts: a `java.net.URLStreamHandler` and a `java.net.URLConnection`. These are both `abstract` classes that we will subclass to create our protocol handler. (Note that these are `abstract` classes, not interfaces. Although they contain abstract methods we are required to implement, they also contain many utility methods we can use or override.) The URL looks up an appropriate `URLStreamHandler`, based on the protocol component of the URL. The `URLStreamHandler` then finishes parsing the URL and creates a `URLConnection` when it's time to communicate with the server. The `URLConnection` represents a single connection with a server and implements the communication protocol itself.

Locating Protocol Handlers

Protocol handlers are organized in a package hierarchy similar to content handlers. But unlike content handlers, which are grouped into packages by the MIME types of the objects that they handle, protocol handlers are given individual packages. Both parts of the protocol handler (the `URLStreamHandler` class and the `URLConnection` class) are located in a package named for the protocol they support.

For example, if we wrote an FTP protocol handler, we might put it in an `learningjava.protocolhandlers.ftp` package. The `URLStreamHandler` is placed in this package and given the name `Handler`; all `URLStreamHandlers` are named `Handler` and distinguished by the package in which they reside. The `URLConnection` portion of the protocol handler is placed in the same package and can be given any name. There is no need for a naming convention because the corresponding `URLStreamHandler` is responsible for creating the `URLConnection` objects it uses.

As with content handlers, Java locates packages containing protocol handlers using the `java.protocol.handler.pkgs` system property. The value of this property is a list of package names; if more than one package is in the list, use a vertical bar (|) to separate them. For our example, we will set this property to include `learningjava.protocolhandlers`.

URLs, Stream Handlers, and Connections

The `URL`, `URLStreamHandler`, `URLConnection`, and `ContentHandler` classes work together closely. Before diving into an example, let's take a step back, look at the parts a little more, and see how these things communicate. Figure A-2 shows how these components relate to each other.

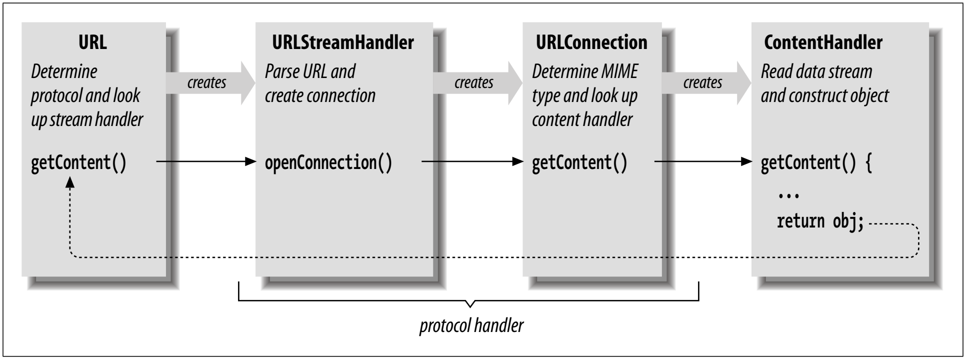

URL
Determine protocol and look up stream handler
getContent()
creates
URLStreamHandler
Parse URL and create connection
openConnection()
creates
URLConnection
Determine MIME type and look up content handler
getContent()
creates
ContentHandler
Read data stream and construct object
getContent() {
...
return obj;
protocol handler

Figure A-2. The protocol handler machinery

We begin with the `URL` object, which points to the resource we'd like to retrieve. The `URLStreamHandler` helps the `URL` class parse the URL specification string for its particular protocol. For example, consider the following call to the `URL` constructor:

```
URL url = new URL("protocol://foo.bar.com/file.ext");
```

The `URL` class parses only the protocol component; later, a call to the `URL` class's `getContent( )` or `openStream( )` method starts the machinery in motion. The `URL` class locates the appropriate protocol handler by looking in the protocol-package hierarchy. It then creates an instance of the appropriate `URLStreamHandler` class.

The `URLStreamHandler` is responsible for parsing the rest of the URL string, including hostname and filename, and possibly an alternative port designation. This allows different protocols to have their own variations on the format of the URL specification string. Note that this step is skipped when a URL is constructed with the "protocol," "host," and "file" components specified explicitly. If the protocol is straightforward, its `URLStreamHandler` class can let Java do the parsing and accept the default behavior. For this illustration, we'll assume that the `URL` string requires no special parsing. (If we use a nonstandard URL with a strange format, we're responsible for parsing it ourselves, as we'll show shortly.)

The `URL` object next invokes the handler's `openConnection( )` method, prompting the handler to create a new `URLConnection` to the resource. The `URLConnection` performs whatever communications are necessary to talk to the resource and begins to fetch data for the object. At that time, it also determines the MIME type of the incoming object data and prepares an `InputStream` to hand to the appropriate content handler. This `InputStream` must send “pure” data with all traces of the protocol removed.

The `URLConnection` also locates an appropriate content handler in the content-handler package hierarchy. The `URLConnection` creates an instance of a content handler; to put the content handler to work, the `URLConnection`’s `getContent( )` method calls the content handler’s `getContent( )` method. If this sounds confusing, it is: we have three `getContent( )` methods calling each other in a chain. The newly created `ContentHandler` object then acquires the stream of incoming data for the object by calling the `URLConnection`’s `getInputStream( )` method. (Recall that we acquired an `InputStream` in our `x_tar` content handler.) The content handler reads the stream and constructs an object from the data. This object is then returned up the `getContent( )` chain: from the content handler, the `URLConnection`, and finally the URL itself. Now our application has the desired object in its greedy little hands.

To summarize, we create a protocol handler by implementing a `URLStreamHandler` class that creates specialized `URLConnection` objects to handle our protocol. The `URLConnection` objects implement the `getInputStream( )` method, which provides data to a content handler for construction of an object. The base `URLConnection` class implements many of the methods we need; therefore, our `URLConnection` needs to provide only the methods that generate the data stream and return the MIME type of the object data.

If you’re not thoroughly confused by all that terminology (or even if you are), let’s move on to the example. It should help to pin down what all these classes are doing.

The crypt Handler

In this section, we’ll build a *crypt* protocol handler. It parses URLs of the form:

```
crypt:type://hostname[:port]/location/item
```

type is an identifier that specifies what kind of encryption to use. The protocol itself is a simplified version of HTTP; we’ll implement the `GET` command and no more. We added the *type* identifier to the URL to show how to parse a nonstandard URL specification. Once the handler has figured out the encryption type, it dynamically loads a class that implements the chosen encryption algorithm and uses it to retrieve the data. Obviously, we don’t have room to implement a full-blown public-key encryption algorithm, so we’ll use the `rot13InputStream` class from Chapter 11. It should be apparent how the example can be extended by plugging in a more powerful encryption class.

The Encryption class

First, we'll lay out our plug-in encryption class. We'll define an abstract class called `CryptInputStream` that provides some essentials for our plug-in encrypted protocol. From the `CryptInputStream` we'll create a subclass called `rot13CryptInputStream`, that implements our particular kind of encryption:

```
//file: rot13CryptInputStream.java
package learningjava.protocolhandlers.crypt;
import java.io.*;

abstract class CryptInputStream extends InputStream {
    InputStream in;
    OutputStream out;
    abstract public void set( InputStream in, OutputStream out );
} // end of class CryptInputStream

class rot13CryptInputStream extends CryptInputStream {

    public void set( InputStream in, OutputStream out ) {
        this.in = new learningjava.io.rot13InputStream( in );
    }
    public int read( ) throws IOException {
        return in.read( );
    }
}
```

Our `CryptInputStream` class defines a method called `set( )` that passes in the `InputStream` it's to translate. Our `URLConnection` calls `set( )` after creating an instance of the encryption class. We need a `set( )` method because we want to load the encryption class dynamically, and we aren't allowed to pass arguments to the constructor of a class when it's dynamically loaded. (We noticed this same issue in our content handler previously.) In the encryption class, we also provide for the possibility of an `OutputStream`. A more complex kind of encryption might use the `OutputStream` to transfer public-key information. Needless to say, rot13 doesn't, so we'll ignore the `OutputStream` here.

The implementation of `rot13CryptInputStream` is very simple. `set( )` takes the `InputStream` it receives and wraps it with the `rot13InputStream` filter. `read( )` reads filtered data from the `InputStream`, throwing an exception if `set( )` hasn't been called.

The URLStreamHandler

Next we'll build our `URLStreamHandler` class. The class name is `Handler`; it extends the abstract `URLStreamHandler` class. This is the class the Java `URL` looks up by converting the protocol name (*crypt*) into a package name. Remember that Java expects this class to be named `Handler`, and to live in a package named for the protocol type.

```
//file: Handler.java
package learningjava.protocolhandlers.crypt;
import java.io.*;
import java.net.*;

public class Handler extends URLStreamHandler {

    protected void parseURL(URL url, String spec,
                            int start, int end) {
        int slash = spec.indexOf('/');
        String crypType = spec.substring(start, slash-1);
        super.parseURL(url, spec, slash, end);
        setURL( url, "crypt:"+crypType, url.getHost( ),
            url.getPort(), url.getFile( ), url.getRef( ) );
    }

    protected URLConnection openConnection(URL url)
      throws IOException {
        String crypType = url.getProtocol( ).substring(6);
        return new CryptURLConnection( url, crypType );
    }
}
```

Java creates an instance of our `URLStreamHandler` when we create a `URL` specifying the *crypt* protocol. `Handler` has two jobs: to assist in parsing the URL specification strings and to create `CryptURLConnection` objects when it's time to open a connection to the host.

Our `parseURL( )` method overrides the `parseURL( )` method in the `URLStreamHandler` class. It's called whenever the `URL` constructor sees a URL requesting the *crypt* protocol. For example:

```
URL url = new URL("crypt:rot13://foo.bar.com/file.txt");
```

`parseURL( )` is passed a reference to the `URL` object, the URL specification string, and starting and ending indexes that show what portion of the URL string we're expected to parse. The `URL` class has already identified the simple protocol name; otherwise, it wouldn't have found our protocol handler. Our version of `parseURL( )` retrieves our *type* identifier from the specification and stores it temporarily in the variable `crypType`. To find the encryption type, we take everything between the starting index we were given and the character preceding the first slash in the URL string (i.e., everything up to the colon in ://). We then defer to the superclass `parseURL( )` method to complete the job of parsing the URL after that point. We call `super.parseURL( )` with the new start index, so that it points to the character just after the type specifier. This tells the superclass `parseURL( )` that we've already parsed everything prior to the first slash, and it's responsible for the rest. Finally we use the utility method `setURL( )` to put together the final URL. Almost everything has already been set correctly for us, but we need to call `setURL( )` to add our special type to the protocol identifier. We'll need this information later when someone wants to open the URL connection.

Before going on, we'll note two other possibilities. If we hadn't hacked the URL string for our own purposes by adding a type specifier, we'd be dealing with a standard URL specification. In this case, we wouldn't need to override parseURL( ); the default implementation would have been sufficient. It could have sliced the URL into host, port, and filename components normally. On the other hand, if we had created a completely bizarre URL format, we would need to parse the entire string. There would be no point calling super.parseURL( ); instead, we'd have called the URLStreamHandler's protected method setURL( ) to pass the URL's components back to the URL object.

The other method in our Handler class is openConnection( ). After the URL has been completely parsed, the URL object calls openConnection( ) to set up the data transfer. openConnection( ) calls the constructor for our URLConnection with appropriate arguments. In this case, our URLConnection object is named CryptURLConnection, and the constructor requires the URL and the encryption type as arguments. parseURL( ) put the encryption type in the protocol identifier of the URL. We recognize it and pass the information along. openConnection( ) returns the reference to our URLConnection, which the URL object uses to drive the rest of the process.

The URLconnection

Finally, we reach the real guts of our protocol handler, the URLConnection class. This is the class that opens the socket, talks to the server on the remote host, and implements the protocol itself. This class doesn't have to be public, so you can put it in the same file as the Handler class we just defined. We call our class CryptURLConnection; it extends the abstract URLConnection class. Unlike ContentHandler and StreamURLConnection, whose names are defined by convention, we can call this class anything we want; the only class that needs to know about the URLConnection is the URLStreamHandler, which we wrote ourselves:

```
//file: CryptURLConnection.java
  import java.io.*;
  import java.net.*;

class CryptURLConnection extends URLConnection {
    static int defaultPort = 80;
    CryptInputStream cis;

    public String getContentType( ) {
        return guessContentTypeFromName( url.getFile( ) );
    }

    CryptURLConnection ( URL url, String crypType )
      throws IOException {
        super( url );
        try {
            String classname = "learningjava.protocolhandlers.crypt."
                + crypType + "CryptInputStream";
            cis = (CryptInputStream)
                   Class.forName(classname).newInstance( );
        } catch ( Exception e ) {
            throw new IOException("Crypt Class Not Found: "+e);
        }
    }

    public void connect( ) throws IOException {
        int port = ( url.getPort( ) == -1 ) ?
                     defaultPort : url.getPort( );
        Socket s = new Socket( url.getHost( ), port );

        // Send the filename in plaintext
        OutputStream server = s.getOutputStream( );
        new PrintWriter( new OutputStreamWriter( server, "8859_1" ),
                         true).println( "GET " + url.getFile( ) );

        // Initialize the CryptInputStream
        cis.set( s.getInputStream( ), server );
        connected = true;
    }

    public InputStream getInputStream( ) throws IOException {
        if (!connected)
            connect( );
        return ( cis );
    }
}
```

The constructor for our CryptURLConnection class takes as arguments the destination URL and the name of an encryption type. We pass the URL on to the constructor of our superclass, which saves it in a protected url instance variable. We could have saved the URL ourselves but calling our parent's constructor shields us from possible changes or enhancements to the base class. We use crypType to construct the name of an encryption class, using the convention that the encryption class is in the same package as the protocol handler (i.e., learningjava.protocolhandlers.crypt); its name is the encryption type followed by the suffix CryptInputStream.

Once we have a name, we need to create an instance of the encryption class. To do so, we use the static method Class.forName( ) to turn the name into a Class object and newInstance( ) to load and instantiate the class. (This is how Java loads the content and protocol handlers themselves.) newInstance( ) returns an Object; we need to cast it to something more specific before we can work with it. Therefore, we cast it to our CryptInputStream class, the abstract class that rot13CryptInputStream extends. If we implement any additional encryption types as extensions to CryptInputStream and name them appropriately, they will fit into our protocol handler without modification.

We do the rest of our setup in the connect( ) method of the URLConnection. There, we make sure we have an encryption class and open a Socket to the appropriate port on the remote host. getPort( ) returns -1 if the URL doesn't specify a port explicitly; in that case we use the default port for an HTTP connection (port 80). We ask for an OutputStream on the socket, assemble a GET command using the getFile( ) method to discover the filename specified by the URL, and send our request by writing it into the OutputStream. (For convenience, we wrap the OutputStream with a PrintWriter and call println( ) to send the message.) We then initialize the CryptInputStream class by calling its set( ) method and passing it an InputStream from the Socket and the OutputStream.

The last thing connect( ) does is set the boolean variable connected to true. connected is a protected variable inherited from the URLConnection class. We need to track the state of our connection because connect( ) is a public method. It's called by the URLConnection's getInputStream( ) method, but it could also be called by other classes. Since we don't want to start a connection if one already exists, we check connected first.

In a more sophisticated protocol handler, connect( ) would also be responsible for dealing with any protocol headers that come back from the server. In particular, it would probably stash any important information it deduced from the headers (e.g., MIME type, content length, time stamp) in instance variables, where it's available to other methods. At a minimum, connect( ) strips the headers from the data so the content handler won't see them. We'll be lazy and assume we'll connect to a minimal server, such as the modified TinyHttpd daemon (discussed in the next section), which doesn't bother with any headers.

The bulk of the work has been done; a few details remain. The URLConnection's getContent( ) method needs to figure out which content handler to invoke for this URL. In order to compute the content handler's name, getContent( ) needs to know the resource's MIME type. To find out, it calls the URLConnection's getContentType( ) method, which returns the MIME type as a String. Our protocol handler overrides getContentType( ), providing our own implementation.

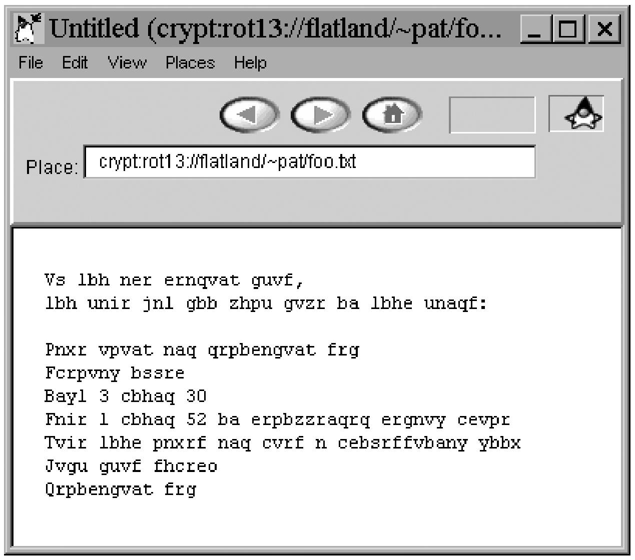
The URLConnection class provides a number of tools to help determine the MIME type. It's possible that the MIME type is conveyed explicitly in a protocol header; in this case, a more sophisticated version of connect( ) would have stored the MIME type in a convenient location for us. Some servers don't bother to insert the appropriate headers, though, so you can use the method guess-ContentTypeFromName( ) to examine filename extensions, like *.gif* or *.html*, and map them to MIME types. In the worst case, you can use guessContent-TypeFromStream( ) to intuit the MIME type from the raw data. The Java developers call this method "a disgusting hack" that shouldn't be needed, but that is unfortunately necessary in a world where HTTP servers lie about content types and extensions are often nonstandard. We'll take the easy way out and use the guessContentTypeFromName( ) utility of the URLConnection class to determine the MIME type from the filename extension of the URL we are retrieving.

Once the URLConnection has found a content handler, it calls the content handler's getContent( ) method. The content handler then needs to get an InputStream from which to read the data. To find an InputStream, it calls the URLConnection's getInputStream( ) method. getInputStream( ) returns an InputStream from which its caller can read the data after protocol processing is finished. It checks whether a connection is already established; if not, it calls connect( ) to make the connection. Then it returns a reference to our CryptInputStream.

A final note on getting the content type: the URLConnection's default getContentType( ) calls getHeaderField( ), which is presumably supposed to extract the named field from the protocol headers (it would probably spit back information connect( ) had stored away). But the default implementation of getHeaderField( ) just returns null; we would have to override it to make it do anything interesting. Several other connection attributes use this mechanism, so in a more general implementation, we'd probably override getHeaderField( ) rather than getContentType( )directly.

Trying it out

Let's try out our new protocol! Compile all the classes and put them in the learningjava.protocolhandlers package somewhere in your class path. Now set the java.protocol.handler.pkgs system property in HotJava to include learningjava.protocolhandlers. Type a "crypt" style URL for a text document; you should see something like that shown in Figure A-3.


Untitled (crypt:rot13://flatland/~pat/fo...
File
Edit
View
Places
Help
Place:
crypt:rot13://flatland/~pat/foo.txt

```
Vs lbh ner ernqvat guvf,
lbh unir jnl gbb zhpu gvzr ba lbhe unaqf:

Pnxr vpvat naq qrpbengvat frg
Fcrpvny bssre
Bayl 3 cbhaq 30
Fnir 1 cbhaq 52 ba erpbzzraqrq ergnvy cevpr
Tvir lbhe pnxrf naq cvrf n cebsrffvbany ybbx
Jvgu guvf fhcreo
Qrpbengvat frg
```

Figure A-3. The crypt protocol handler at work

This example would be more interesting if we had a rot13 server. Since the *crypt* protocol is nothing more than HTTP with some encryption added, we can make a rot13 server by modifying one line of the TinyHttpd server we developed in Chapter 12, so that it spews its files in rot13. Just change the line that reads the data from the file—replace this line:

```
f.read( data );
```

with a line that reads through a `rot13InputStream`:

```
new learningjava.io.rot13InputStream( f ).read( data );
```

We'll assume you placed the `rot13InputStream` example in a package called `learningjava.io`, and that it's somewhere in your class path. Now recompile and run the server. It automatically encodes the files before sending them; our sample application decodes them on the other end.

We hope that this example has given you some food for thought. Content and protocol handlers are among the most exciting ideas in Java. It's unfortunate that we have to wait for future releases of HotJava and Netscape to take full advantage of them. But in the meantime, you can experiment and implement your own applications.