Table of Contents

Chapter 16. Protocol Handlers

When designing an architecture that would allow them to build a self-extensible browser, the engineers at Sun divided the problem into two parts: handling protocols and handling content. Handling a protocol involves the interaction between a client and a server: generating requests in the correct format, interpreting the headers that come back with the data, acknowledging that the data has been received, etc. Handling the content involves converting the raw data into a format Java understands—for example, an `InputStream` or an `AudioClip`. These two problems, handling protocols and handling content, are distinct. The software that displays a GIF image doesn't care whether the image was retrieved via FTP, HTTP, gopher, or some new protocol. Likewise, the protocol handler, which manages the connection and interacts with the server, doesn't care if it's receiving an HTML file or an MPEG movie file; at most, it will extract a content type from the headers to pass along to the content handler.

Java divides the task of handling protocols into a number of pieces. As a result, there is no single class called `ProtocolHandler`. Instead, four different classes in the `java.net` package work together to implement the protocol handler mechanism. Those classes are `URL`, `URLStreamHandler`, `URLConnection`, and `URLStreamHandlerFactory`. `URL` is the only concrete class in this group; `URLStreamHandler` and `URLConnection` are abstract classes and `URLStreamHandlerFactory` is an interface. Therefore, if you are going to implement a new protocol handler, you have to write concrete subclasses for the `URLStreamHandler` and the `URLConnection`. To use these classes, you may also have to write a class that implements the `URLStreamHandlerFactory` interface.

16.1. What Is a Protocol Handler?

The way the `URL`, `URLStreamHandler`, `URLConnection`, and `URLStreamHandlerFactory` classes work together can be confusing. Everything starts with a URL, which represents a pointer to a particular Internet resource. Each URL specifies the protocol used to access the resource; typical values for the protocol include `mailto`, `http`, and `ftp`. When you construct a `URL` object from the URL's string representation, the constructor strips the protocol field and passes it to the `URLStreamHandlerFactory`. The factory's job is to take the protocol, locate the right subclass of `URLStreamHandler` for the protocol, and create a new instance of that stream handler, which is stored as a field within the `URL` object. Each application has at most one `URLStreamHandlerFactory`; once the factory has been installed, attempting to install another will throw an `Error`.

Now that the `URL` object has a stream handler, it asks the stream handler to finish parsing the URL string and create a subclass of `URLConnection` that knows how to talk to servers using this protocol. `URLStreamHandler` subclasses and `URLConnection` subclasses always come in pairs; the stream handler for a protocol always knows how to find an appropriate `URLConnection` for its protocol. It is worth noting that the stream handler does most of the work of parsing the URL. The format of the URL, although standard, depends on the protocol; therefore, it must be parsed by a `URLStreamHandler`, which knows about a particular protocol, and not by the `URL` object, which is generic and has no knowledge of specific protocols. This also means that if you are writing a new stream handler, you can define a new URL format that's appropriate to your task.

New URL schemes should be defined only for genuinely new protocols. They should not be defined for different uses of existing protocols. The iTunes Music Store *itms* scheme and the RSS *feed* scheme are examples of what not to do. Both of these should use *http*.

The `URLConnection` class, which you learned about in the previous chapter, represents an active connection to an Internet resource. It is responsible for interacting with the server. A `URLConnection` knows how to generate requests and interpret the headers that the server returns. The output from a `URLConnection` is the raw data requested with all traces of the protocol (headers, etc.) stripped, ready for processing by a content handler.

In most applications, you don't need to worry about `URLConnection` objects and stream handlers; they are hidden by the `URL` class, which provides a simple interface to the functionality you need. When you call the `getInputStream( )`, `getOutputStream ()`, and `getContent( )` methods of the `URL` class, you are really calling similarly named methods in the `URLConnection` class. We have seen that interacting directly with a `URLConnection` can be convenient when you need a little more control over communication with a server, such as when downloading binary files or posting data to a server-side program.

However, the `URLConnection` and `URLStreamHandler` classes are even more important when you need to add new protocols. By writing subclasses of these classes, you can add support for standard protocols such as finger, whois, or NTP that Java doesn't support out of the box. Furthermore, you're not limited to established protocols with well-known services. You can create new protocols that perform database queries, search across multiple Internet search engines, view pictures from binary newsgroups, and more. You can add new kinds of URLs as needed to represent the new types of resources. Furthermore, Java applications can be built so that they load new protocol handlers at runtime. Unlike current browsers such as Mozilla and Internet Explorer, which contain explicit knowledge of all the protocols and content types they can handle, a Java browser can be a relatively lightweight skeleton that loads new handlers as needed. Supporting a new protocol just means adding some new classes in predefined locations, not writing an entirely new release of the browser.

What's involved in adding support for a new protocol? As I said earlier, you need to write two new classes: a subclass of `URLConnection` and a subclass of `URLStreamHandler`. You may also need to write a class that implements the `URLStreamHandlerFactory` interface. The `URLConnection` subclass handles the interaction with the server, converts anything the server sends into an `InputStream`, and converts anything the client sends into an `OutputStream`. This subclass must implement the abstract method `connect( )`; it may also override the concrete methods `getInputStream( )`, `getOutputStream( )`, and `getContentType( )`.

The `URLStreamHandler` subclass parses the string representation of the URL into its separate parts and creates a new `URLConnection` object that understands that URL's protocol. This subclass must implement the abstract `openConnection( )` method, which returns the new `URLConnection` to its caller. If the `String` representation of the URL doesn't look like a standard hierarchical URL, you should also override the `parseURL( )` and `toExternalForm( )` methods.

Finally, you may need to create a class that implements the `URLStreamHandlerFactory` interface. The `URLStreamHandlerFactory` helps the application find the right protocol handler for each type of URL. The `URLStreamHandlerFactory` interface has a single method, `createURLStreamHandler( )`, which returns a `URLStreamHandler` object. This method must find the appropriate subclass of `URLStreamHandler` given only the protocol (e.g., *ftp*); that is, it must understand the package and class-naming conventions used for stream handlers. Since `URLStreamHandlerFactory` is an interface, you can place the `createURLStreamHandler( )` method in any convenient class, perhaps the main class of your application.

When it first encounters a protocol, Java looks for `URLStreamHandler` classes in this order:

1. First, Java checks to see whether a `URLStreamHandlerFactory` is installed. If it is, the factory is asked for a `URLStreamHandler` for the protocol.
2. If a `URLStreamHandlerFactory` isn't installed or if Java can't find a `URLStreamHandler` for the protocol, Java looks in the packages named in the `java.protocol.handler.pkgs` system property for a sub-package that shares the protocol name and a class called `Handler`. The value of this property is a list of package names separated by a vertical bar (`|`). Thus, to indicate that Java should seek protocol handlers in the `com.macfaq.net.www` and `org.cafeaulait.protocols` packages, you would add this line to your properties file:

   ```
   java.protocol.handler.pkgs=com.macfaq.net.www|org.cafeaulait.protocols
   ```

   To find an FTP protocol handler (for example), Java first looks for the class `com.macfaq.net.www.ftp.Handler`. If that's not found, Java next tries to instantiate `org.cafeaulait.protocols.ftp.Handler`.
3. Finally, if all else fails, Java looks for a `URLStreamHandler` named `sun.net.www.protocol.`*`name`*`.Handler`, where *`name`* is replaced by the name of the protocol; for example, `sun.net.www.protocol.ftp.Handler`.

In the early days of Java (circa 1995), Sun promised that protocols could be installed at runtime from the server that used them. For instance, in 1996, James Gosling and Henry McGilton wrote: "The HotJava Browser is given a reference to an object (a URL). If the handler for that protocol is already loaded, it will be used. If not, the HotJava Browser will search first the local system and then the system that is the target of the URL." (*The Java Language Environment, A White Paper*, May 1996, http://java.sun.com/docs/white/langenv/HotJava.doc1.html) However, the loading of protocol handlers from web sites was never implemented, and Sun doesn't talk much about it anymore.

Most of the time, an end user who wants to permanently install an extra protocol handler in a program such as HotJava will place the necessary classes in the program's class path and add the package prefix to the `java.protocol.handler.pkgs` property. However, a programmer who just wants to add a custom protocol handler to their program at compile time will write and install a `URLStreamHandlerFactory` that knows how to find their custom protocol handlers. The factory can tell an application to look for `URLStreamHandler` classes in any place that's convenient: on a web site, in the same directory as the application, or somewhere in the user's class path.

When each of these classes has been written and compiled, you're ready to write an application that uses the new protocol handler. Assuming that you're using a `URLStreamHandlerFactory`, pass the factory object to the static `URL.setURLStreamHandlerFactory()` method like this:

```
URL.setURLStreamHandlerFactory(new MyURLStreamHandlerFactory( ));
```

This method can be called only once in the lifetime of an application. If it is called a second time, it will throw an `Error`. Untrusted code will generally not be allowed to install factories or change the `java.protocol.handler.pkgs` property. Consequently, protocol handlers are primarily of use to standalone applications such as HotJava; Netscape and

Internet Explorer use their own native C code instead of Java to handle protocols, so they're limited to a fixed set of protocols.

To summarize, here's the sequence of events:

1. The program constructs a `URL` object.
2. The constructor uses the arguments it's passed to determine the protocol part of the URL, e.g., *http*.
3. The `URL( )` constructor tries to find a `URLStreamHandler` for the given protocol like this:
    a. If the protocol has been used before, the `URLStreamHandler` object is retrieved from a cache.
    b. Otherwise, if a `URLStreamHandlerFactory` has been set, the protocol string is passed to the factory's `createURLStreamHandler( )` method.
    c. If the protocol hasn't been seen before and there's no `URLStreamHandlerFactory`, the constructor attempts to instantiate a `URLStreamHandler` object named *`protocol`*`.Handler` in one of the packages listed in the `java.protocol.handler.pkgs` property.
    d. Failing that, the constructor attempts to instantiate a `URLStreamHandler` object named *`protocol`*`.Handler` in the `sun.net.www.protocol` package.
    e. If any of these attempts succeed in retrieving a `URLStreamHandler` object, the `URL` constructor sets the `URL` object's `handler` field. If none of the attempts succeed, the constructor throws a `MalformedURLException`.
4. The program calls the `URL` object's `openConnection( )` method.
5. The `URL` object asks the `URLStreamHandler` to return a `URLConnection` object appropriate for this URL. If there's any problem, an `IOException` is thrown. Otherwise, a `URLConnection` object is returned.
6. The program uses the methods of the `URLConnection` class to interact with the remote resource.

Instead of calling `openConnection( )` in step 4, the program can call `getContent( )` or `getInputStream( )`. In this case, the `URLStreamHandler` still instantiates a `URLConnection` object of the appropriate class. However, instead of returning the `URLConnection` object itself, the `URLStreamHandler` returns the result of `URLConnection`'s `getContent( )` or `getInputStream()` method.

16.2. The URLStreamHandler Class

The abstract `URLStreamHandler` class is a superclass for classes that handle specific protocols—for example, HTTP. You rarely call the methods of the `URLStreamHandler` class; they are called by other methods in the `URL` and `URLConnection` classes. By overriding the `URLStreamHandler` methods in your own subclass, you teach the `URL` class how to handle new protocols. Therefore, I'll focus on overriding the methods of `URLStreamHandler` rather than calling the methods.

16.2.1. The Constructor

You do not create `URLStreamHandler` objects directly. Instead, when a URL is constructed with a protocol that hasn't been seen before, Java asks the application's

`URLStreamHandlerFactory` to create the appropriate `URLStreamHandler` subclass for the protocol. If that fails, Java guesses at the fully package-qualified name of the `URLStreamHandler` class and uses `Class.forName( )` to attempt to construct such an object. This means each concrete subclass should have a noargs constructor. The single constructor for `URLStreamHandler` doesn't take any arguments:

```
public URLStreamHandler( )
```

Because `URLStreamHandler` is an abstract class, this constructor is never called directly; it is only called from the constructors of subclasses.

16.2.2. Methods for Parsing URLs

The first responsibility of a `URLStreamHandler` is to split a string representation of a `URL` into its component parts and use those parts to set the various fields of the `URL` object. The `parseURL( )` method splits the URL into parts, possibly using `setURL( )` to assign values to the `URL`'s fields. It is very difficult to imagine a situation in which you would call `parseURL( )` directly; instead, you override it to change the behavior of the `URL` class.

16.2.2.1. protected void parseURL(URL u, String spec, int start, int limit)

This method parses the `String spec` into a `URL u`. All characters in the `spec` string before `start` should already have been parsed into the `URL u`. Characters after `limit` are ignored. Generally, the protocol will have already been parsed and stored in `u` before this method is invoked, and `start` will be adjusted so that it starts with the character after the colon that delimits the protocol.

The task of `parseURL( )` is to set `u`'s `protocol`, `host`, `port`, `file`, and `ref` fields. It can assume that any parts of the `String` that are before `start` and after `limit` have already been parsed or can be ignored.

The `parseURL( )` method that Java supplies assumes that the URL looks more or less like an *http* or other hierarchical URL:

```
protocol://www.host.com:port/directory/another_directory/file#fragmentID
```

This works for *ftp* and *gopher* URLs. It does not work for *mailto* or *news* URLs and may not be appropriate for any new URL schemes you define. If the protocol handler uses URLs that fit this hierarchical form, you don't have to override `parseURL()` at all; the method inherited from URLStreamHandler works just fine. If the URLs are completely different, you must supply a parseURL( ) method that parses the URL completely. However, there's often a middle ground that can make your task easier. If your URL looks somewhat like a standard URL, you can implement a parseURL( ) method that handles the nonstandard portion of the URL and then calls super.parseURL( ) to do the rest of the work, setting the offset and limit arguments to indicate the portion of the URL that you didn't parse.

For example, a *mailto* URL looks like mailto:elharo@metalab.unc.edu. First, you need to figure out how to map this into the URL class's protocol, host, port, file, and ref fields. The protocol is clearly mailto. Everything after the @ can be the host. The hard question is what to do with the username. Since a *mailto* URL really doesn't have a file portion, we will use the URL class's file field to hold the username. The ref can be set to the empty string or null. The parseURL( ) method that follows implements this scheme:

```
public void parseURL(URL u, String spec, int start, int limit) {

  String protocol = u.getProtocol( );
  String host = "";
  int port = u.getPort( );
  String file = ""; // really username
  String fragmentID  = null;

  if( start < limit) {
    String address = spec.substring(start, limit);
    int atSign = address.indexOf('@');
    if (atSign >= 0) {
      host = address.substring(atSign+1);
      file = address.substring(0, atSign);
    }
  }
  this.setURL(u, protocol, host, port, file, fragmentID );
}
```

Rather than borrowing an unused field from the URL object, it's possibly a better idea to store protocol-specific parts of the URL, such as the username, in fields of the URLStreamHandler subclass. The disadvantage of this approach is that such fields can be seen only by your own code; in this example, you couldn't use the getFile( ) method in the URL class to retrieve the username. Here's a version of parseURL( ) that stores the username in a field of the Handler subclass. When the connection is opened, the username can be copied into the MailtoURLConnection object that results. That class would provide some sort of getUserName( ) method:

```
String username = "";

public void parseURL(URL u, String spec, int start, int limit) {

  String protocol = u.getProtocol( );
  String host = "";
```

```
    int port = u.getPort( );
    String file = "";
    String fragmentID  = null;

    if( start < limit) {
      String address = spec.substring(start, limit);
      int atSign = address.indexOf('@');
      if (atSign >= 0) {
        host = address.substring(atSign+1);
        this.username = address.substring(0, atSign);
      }
    }
    this.setURL(u, protocol, host, port, file, fragmentID );

  }
```

16.2.2.2. protected String toExternalForm(URL u)

This method puts the pieces of the `URL` `u`—that is, its `protocol`, `host`, `port`, `file`, and `ref` fields—back together in a `String`. A class that overrides `parseURL()` should also override `toExternalForm( )`. Here's a `toExternalForm()` method for a *mailto* URL; it assumes that the username has been stored in the `URL`'s `file` field:

```
  protected String toExternalForm(URL u) {

    return "mailto:" + u.getFile( ) + "@" + u.getHost( );

  }
```

Since `toExternalForm( )` is protected, you probably won't call this method directly. However, it is called by the public `toExternalForm( )` and `toString( )` methods of the `URL` class, so any change you make here is reflected when you convert `URL` objects to strings.

16.2.2.3. protected void setURL(URL u, String protocol, String host, int port, String authority, String userInfo, String path, String query, String fragmentID) // Java 1.3

This method sets the `protocol`, `host`, `port`, `authority`, `userInfo`, `path`, `query`, and `ref` fields of the `URL` `u` to the given values. `parseURL( )` uses this method to set these fields to the values it has found by parsing the URL. You need to call this method at the end of the `parseURL( )` method when you subclass `URLStreamHandler`.

This method is a little flaky, since the host, port, and user info together make up the authority. In the event of a conflict between them, they're all stored separately, but the host, port, and user info are used in preference to the authority when deciding which site to connect to.

This is actually quite relevant to the *mailto* example, since *mailto* URLs often have query strings that indicate the subject or other header; for example, mailto:elharo@metalab.unc.edu?subject=JavaReading. Here the query string is *subject=JavaReading*. Rewriting the `parseURL( )` method to support *mailto* URLs in this format, the result looks like this:

```
public void parseURL(URL u, String spec, int start, int limit) {

  String protocol    = u.getProtocol( );
  String host        = "";
  int port           = u.getPort( );
  String file        = "";
  String userInfo    = null;
  String query       = null;
  String fragmentID  = null;

  if (start < limit) {
    String address = spec.substring(start, limit);
    int atSign = address.indexOf('@');
    int questionMark = address.indexOf('?');
    int hostEnd = questionMark >= 0 ? questionMark : address.length( );
    if (atSign >= 0) {
      host = address.substring(atSign+1, hostEnd);
      userInfo = address.substring(0, atSign);
    }
    if (questionMark >= 0 && questionMark > atSign) {
      query = address.substring(questionMark + 1);
    }
  }
  String authority = "";
  if (userInfo != null) authority += userInfo + '@';
  authority += host;
  if (port >= 0) authority += ":" + port;

  this.setURL(u, protocol, host, port, authority, userInfo, file,
   query, fragmentID );

}
```

16.2.2.4. protected int getDefaultPort( ) // Java 1.3

The `getDefaultPort()` method returns the default port for the protocol, e.g., 80 for HTTP. The default implementation of this method simply returns -1, but each subclass should override that with the appropriate default port for the protocol it handles. For example, here's a `getDefaultPort()` method for the finger protocol that normally operates on port 79:

```
public int getDefaultPort( ) {
  return 79;
}
```

As well as providing the right port for finger, overriding this method also makes `getDefaultPort( )` public. Although there's only a default implementation of this method in Java 1.3, there's no reason you can't provide it in your own subclasses in any version of Java. You simply won't be able to invoke it polymorphically from a reference typed as the superclass.

16.2.2.5. protected InetAddress getHostAddress(URL u) // Java 1.3

The `getHostAddress()` method returns an `InetAddress` object pointing to the server in the URL. This requires a DNS lookup, and the method does block while the lookup is made. However, it does not throw any exceptions. If the host can't be located, whether because the URL does not contain host information as a result of a DNS failure or a `SecurityException`, it simply returns null. The default implementation of this method is sufficient for any reasonable case. It shouldn't be necessary to override it.

16.2.2.6. protected boolean hostsEqual(URL u1, URL u2) // Java 1.3

The `hostsEqual( )` method determines whether the two URLs refer to the same server. This method does use DNS to look up the hosts. If the DNS lookups succeed, it can tell that, for example, http://www.ibiblio.org/Dave/this-week.html and ftp://metalab.unc.edu/pub/linux/distributions/debian/ are the same host. However, if the DNS lookup fails for any reason, then `hostsEqual( )` falls back to a simple case-insensitive string comparison, in which case it would think these were two different hosts.

The default implementation of this method is sufficient for most cases. You probably won't need to override it. The only case I can imagine where you might want to is if you were trying to make mirror sites on different servers appear equal.

16.2.2.7. protected boolean sameFile(URL u1, URL u2) // Java 1.3

The `sameFile( )` method determines whether two URLs point to the same file. It does this by comparing the protocol, host, port, and path. The files are considered to be the same only if each of those four pieces is the same. However, it does not consider the query string or the fragment identifier. Furthermore, the hosts are compared by the `hostsEqual( )` method so that *www.ibiblio.org* and *metalab.unc.edu* can be recognized as the same if DNS can resolve them. This is similar to the `sameFile()` method of the `URL` class. Indeed, that `sameFile ( )` method just calls this `sameFile( )` method.

The default implementation of this method is sufficient for most cases. You probably won't need to override it. You might perhaps want to do so if you need a more sophisticated test that converts paths to canonical paths or follows redirects before determining whether two URLs have the same file part.

16.2.2.8. protected boolean equals(URL u1, URL u2) // Java 1.3

The final equality method tests almost the entire URL, including protocol, host, file, path, and fragment identifier. Only the query string is ignored. All five of these must be equal for the two URLs to be considered equal. Everything except the fragment identifier is compared by the `sameFile()` method, so overriding that method changes the behavior of this one. The fragment identifiers are compared by simple string equality. Since the `sameFile( )` method uses `hostsEqual( )` to compare hosts, this method does too. Thus, it performs a DNS lookup if possible and may block. The `equals( )` method of the `URL` class calls this method to compare two `URL` objects for equality. Again, you probably won't need to override this method. The default implementation should suffice for most purposes.

16.2.2.9. protected int hashCode(URL u) // Java 1.3

`URLStreamHandler`s can change the default hash code calculation by overriding this method. You should do this if you override `equals( )`, `sameFile()`, or `hostsEqual ( )` to make sure that two equal `URL` objects will have the same hash code, and two unequal `URL` objects will not have the same hash code, at least to a very high degree of probability.

16.2.3. A Method for Connecting

The second responsibility of a `URLStreamHandler` is to create a `URLConnection` object appropriate to the URL. This is done with the abstract `openConnection( )` method.

16.2.3.1. protected abstract URLConnection openConnection(URL u) throws IOException

This method must be overridden in each subclass of `URLConnection`. It takes a single argument, `u`, which is the URL to connect to. It returns an unopened `URLConnection`, directed at the resource `u` points to. Each subclass of `URLStreamHandler` should know how to find the right subclass of `URLConnection` for the protocol it handles.

The `openConnection( )` method is protected, so you usually do not call it directly; it is called by the `openConnection( )` method of a `URL` class. The `URL u` that is passed as an argument is the `URL` that needs a connection. Subclasses override this method to handle a specific protocol. The subclass's `openConnection( )` method is usually extremely simple; in most cases, it just calls the constructor for the appropriate subclass of `URLConnection`. For example, a `URLStreamHandler` for the *mailto* protocol might have an `openConnection( )` method that looks like this:

```
protected URLConnection openConnection(URL u) throws IOException {
  return new com.macfaq.net.www.protocol.MailtoURLConnection(u);
}
```

Example 16-1 demonstrates a complete `URLStreamHandler` for *mailto* URLs. The name of the class is `Handler`, following Sun's naming conventions. It assumes the existence of a `MailtoURLConnection` class.

Example 16-1. A mailto URLStreamHandler

```
package com.macfaq.net.www.protocol.mailto;

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

public class Handler extends URLStreamHandler {

  protected URLConnection openConnection(URL u) throws IOException {
    return new MailtoURLConnection(u);
  }

  public void parseURL(URL u, String spec, int start, int limit) {

    String protocol    = u.getProtocol( );
    String host        = "";
    int    port        = u.getPort( );
    String file        = ""; // really username
    String userInfo    = null;
    String authority   = null;
    String query       = null;
    String fragmentID  = null;

    if( start < limit) {
      String address = spec.substring(start, limit);
      int atSign = address.indexOf('@');
      if (atSign >= 0) {
        host = address.substring(atSign+1);
        file = address.substring(0, atSign);
      }
    }

   // For Java 1.2 comment out this next line
```

```
        this.setURL(u, protocol, host, port, authority,
                      userInfo, file, query, fragmentID );

         // In Java 1.2 and earlier uncomment the following line:
         // this.setURL(u, protocol, host, port, file, fragmentID );

       }

       protected String toExternalForm(URL u) {

         return "mailto:" + u.getFile( ) + "@" + u.getHost( );;

       }
     }
```

16.2.3.2. protected URLConnection openConnection(URL u, Proxy p) throws IOException // Java 1.5

Java 1.5 overloads the `openConnection( )` method to allow you to specify a proxy server for the connection. The `java.net.Proxy` class (also new in Java 1.5) encapsulates the address of a proxy server. Rather than connecting to the host directly, this `URLConnection` connects to the specified proxy server, which relays data back and forth between the client and the server. Protocols that do not support proxies can simply ignore the second argument.

Normally connections are opened with the usual proxy server settings within that VM. Calling this method is only necessary if you want to use a different proxy server. If you want to bypass the usual proxy server and connect directly instead, pass the constant `Proxy.NO_PROXY` as the second argument.

16.3. Writing a Protocol Handler

To demonstrate a complete protocol handler, let's write one for the finger protocol defined in RFC 1288 and introduced in Chapter 9. Finger is a relatively simple protocol compared to JDK-supported protocols such as HTTP and FTP. The client connects to port 79 on the server and sends a list of usernames followed by a carriage return/linefeed pair. The server responds with ASCII text containing information about each of the named users or, if no names are listed, a list of the currently logged in users. For example:

```
% telnet rama.poly.edu 79
Trying 128.238.10.212...
Connected to rama.poly.edu.
Escape character is '^]'.
```

```
Login       Name            TTY    Idle    When    Where
jacola   Jane Colaginae    *pts/7       Tue 08:01  208.34.37.104
marcus   Marcus Tullius     pts/15  13d Tue 17:33  farm-dialup11.poly.e
matewan  Sepin Matewan     *pts/17  17: Thu 15:32  128.238.10.177
hengpi   Heng Pin          *pts/10      Tue 10:36  128.238.18.119
nadats   Nabeel Datsun      pts/12   56 Mon 10:38  128.238.213.227
matewan  Sepin Matewan     *pts/8     4 Sun 18:39  128.238.10.177
Connection closed by foreign host.
```

Or to request information about a specific user:

```
% telnet rama.poly.edu 79
Trying 128.238.10.212...
Connected to rama.poly.edu.
Escape character is '^]'.
marcus
Login       Name            TTY    Idle    When    Where
marcus   Marcus Tullius     pts/15  13d Tue 17:33  farm-dialup11.poly.e
```

Since there's no standard for the format of a finger URL, we will start by creating one. Ideally, this should look as much like an *http* URL as possible. Therefore, we will implement a finger URL like this:

```
finger://hostname:port/usernames
```

Second, we need to determine the content type returned by the finger protocol's `getContentType()` method. New protocols such as HTTP use MIME headers to indicate the content type; in these cases, you do not need to override the default `getContentType( )` method provided by the `URLConnection` class. However, since most protocols precede MIME, you often need to specify the MIME type explicitly or use the static methods `URLConnection.guessContentTypeFromName(String name)` and `URLConnection.guessContentTypeFromStream(InputStream in)` to make an educated guess. This example doesn't need anything so complicated, however. A finger server returns ASCII text, so the `getContentType()` method should return the string `text/plain`. The `text/plain` MIME type has the advantage that Java already understands it. In the next chapter, you'll learn how to write content handlers that let Java understand additional MIME types.

Example 16-2 is a `FingerURLConnection` class that subclasses `URLConnection`. This class overrides the `getContentType( )` and `getInputStream( )` methods of `URLConnection` and implements `connect( )`. It also has a constructor that builds a new `URLConnection` from a URL.

Example 16-2. The FingerURLConnection class

```
package com.macfaq.net.www.protocol.finger;

import java.net.*;
import java.io.*;

public class FingerURLConnection extends URLConnection {

  private Socket connection = null;

  public final static int DEFAULT_PORT = 79;

  public FingerURLConnection(URL u) {
    super(u);
  }

  public synchronized InputStream getInputStream( ) throws IOException {

    if (!connected) this.connect( );
    InputStream in = this.connection.getInputStream( );
    return in;

  }

  public String getContentType( ) {
    return "text/plain";
  }
  public synchronized void connect( ) throws IOException {

    if (!connected) {
      int port = url.getPort( );
      if ( port < 1 || port > 65535) {
        port = DEFAULT_PORT;
      }
      this.connection = new Socket(url.getHost( ), port);
      OutputStream out = this.connection.getOutputStream( );
      String names = url.getFile( );
      if (names != null && !names.equals("")) {
        // delete initial /
        names = names.substring(1);
        names = URLDecoder.decode(names);
        byte[] result;
        try {
          result = names.getBytes("ASCII");
        }
        catch (UnsupportedEncodingException ex) {
          result = names.getBytes( );
        }
        out.write(result);
      }
      out.write('\r');
      out.write('\n');
      out.flush( );
      this.connected = true;
    }
```

```
        }
    }
```

This class has two fields. `connection` is a `Socket` between the client and the server. Both the `getInputStream( )` method and the `connect()` method need access to this field, so it can't be a local variable. The second field is `DEFAULT_PORT`, a `final static int`, which contains the finger protocol's default port; this port is used if the URL does not specify the port explicitly.

The class's constructor holds no surprises. It just calls the superclass's constructor with the same argument, the `URL u`. The `connect( )` method opens a connection to the specified server on the specified port or, if no port is specified, to the default finger port, 79. It sends the necessary request to the finger server. If any usernames were specified in the file part of the URL, they're sent. Otherwise, a blank line is sent. Assuming the connection is successfully opened (no exception is thrown), it sets the `boolean` field `connected` to `true`. Recall from the previous chapter that `connected` is a protected field in `java.net.URLConnection`, which is inherited by this subclass. The `Socket` that `connect( )` opens is stored in the field `connection` for later use by `getInputStream ( )`. The `connect( )` and `getInputStream( )` methods are synchronized to avoid a possible race condition on the `connected` variable.

The `getContentType( )` method returns a `String` containing a MIME type for the data. This is used by the `getContent( )` method of `java.net.URLConnection` to select the appropriate content handler. The data returned by a finger server is almost always ASCII text or some reasonable approximation thereof, so this `getContentType( )` method always returns `text/plain`. The `getInputStream( )` method returns an `InputStream`, which it gets from the `Socket` that `connect` created. If the connection has not already been established when `getInputStream( )` is called, the method calls `connect( )` itself.

Once you have a `URLConnection`, you need a subclass of `URLStreamHandler` that knows how to handle a finger server. This class needs an `openConnection()` method that builds a new `FingerURLConnection` from a URL. Since we defined the *finger* URL as a hierarchical URL, we don't need to implement a `parseURL()` method. Example 16-3 is a stream handler for the finger protocol. For the moment, we're going to use Sun's convention for naming protocol handlers; we call this class `Handler` and place it in the package `com.macfaq.net.www.protocol.finger`.

Example 16-3. The finger handler class

```
package com.macfaq.net.www.protocol.finger;

import java.net.*;
import java.io.*;

public class Handler extends URLStreamHandler {

  public int getDefaultPort( ) {
    return 79;
  }

  protected URLConnection openConnection(URL u) throws IOException {
    return new FingerURLConnection(u);
  }

}
```

You can use HotJava to test this protocol handler. Add the following line to your *.hotjava/properties* file or some other place from which HotJava will load it:

```
java.protocol.handler.pkgs=com.macfaq.net.www.protocol
```

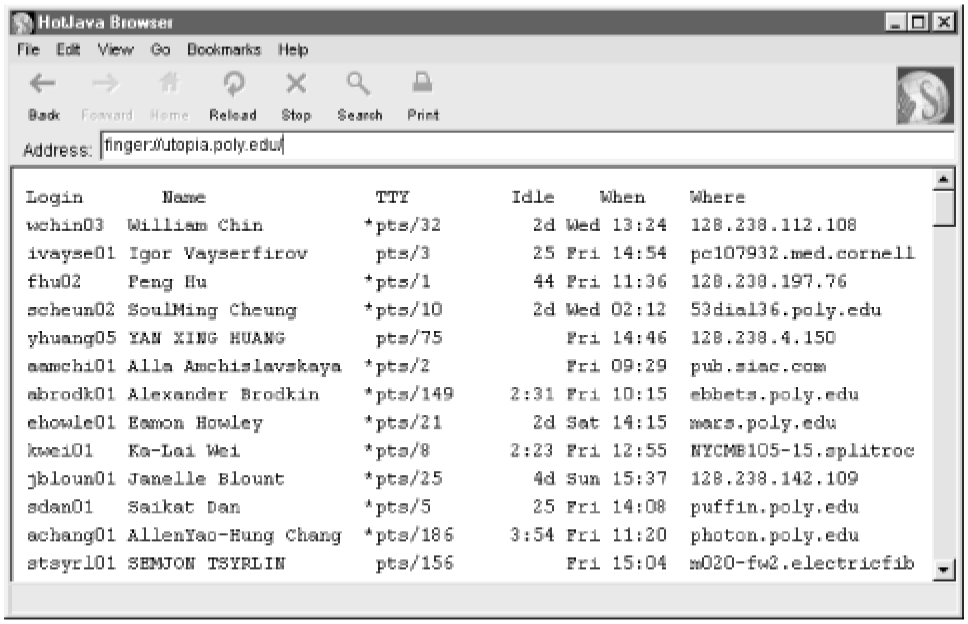
Some (but not all) versions of HotJava may also allow you to set the property from the command line:

```
% hotjava -Djava.protocol.handler.pkgs=com.macfaq.net.www.protocol
```

You also need to make sure that your classes are somewhere in HotJava's class path. HotJava does not normally use the CLASSPATH environment variable to look for classes, so just putting them someplace where the JDK or JRE can find them may not be sufficient. Using HotJava 3.0 on Windows with the JDK 1.3, I was able to put my classes in the *jdk1.3/jre/lib/classes* folder. Your mileage may vary depending on the version of HotJava you're using with which version of the JDK on which platform.

Run it and ask for a URL of a site running finger, such as *utopia.poly.edu*. Figure 16-1 shows the result.

Figure 16-1. HotJava using the finger protocol handler


HotJava Browser
File Edit View Go Bookmarks Help
Back Forward Home Reload Stop Search Print
Address: finger://utopia.poly.edu/

```
Login      Name                 TTY        Idle    When     Where
wchin03  William Chin          *pts/32       2d Wed 13:24  128.238.112.108
ivayse01 Igor Vayserfirov       pts/3        25 Fri 14:54  pc107932.med.cornell
fhu02    Peng Hu               *pts/1        44 Fri 11:36  128.238.197.76
scheun02 SoulMing Cheung       *pts/10       2d Wed 02:12  53dial36.poly.edu
yhuang05 YAN XING HUANG         pts/75          Fri 14:46  128.238.4.150
aamchi01 Alla Amchislavskaya   *pts/2           Fri 09:29  pub.siac.com
abrodk01 Alexander Brodkin     *pts/149    2:31 Fri 10:15  ebbets.poly.edu
ehowle01 Eamon Howley          *pts/21       2d Sat 14:15  mars.poly.edu
kwei01   Ka-Lai Wei            *pts/8      2:23 Fri 12:55  NYCMB105-15.splitroc
jbloun01 Janelle Blount        *pts/25       4d Sun 15:37  128.238.142.109
sdan01   Saikat Dan            *pts/5        25 Fri 14:08  puffin.poly.edu
achang01 AllenYao-Hung Chang   *pts/186    3:54 Fri 11:20  photon.poly.edu
stsyrl01 SEMJON TSYRLIN         pts/156         Fri 15:04  m020-fw2.electricfib
```

16.4. More Protocol Handler Examples and Techniques

Now that you've seen how to write one protocol handler, it's not at all difficult to write more. Remember the five basic steps of creating a new protocol handler:

1. Design a URL for the protocol if a standard URL for that protocol doesn't already exist. As of mid-2004, the official list of URL schemes at the IANA (http://www.iana.org/assignments/uri-schemes) includes only 43 different URL schemes and reserves three more. For anything else, you need to define your own.
2. Decide what MIME type should be returned by the protocol handler's `getContentType( )` method. The text/plain content type is often appropriate for legacy protocols. Another option is to convert the incoming data to HTML inside `getInputStream( )` and return text/html. Binary data often uses one of the many application types. In some cases, you may be able to use the `URLConnection.guessContentTypeFromName( )` or `URLConnection.guessContentTypeFromStream( )` methods to determine the right MIME type.
3. Write a subclass of `URLConnection` that understands this protocol. It should implement the `connect( )` method and may override the `getContentType( )`, `getOutputStream( )`, and `getInputStream( )` methods of `URLConnection`. It also needs a constructor that builds a new `URLConnection` from a `URL`.
4. Write a subclass of `URLStreamHandler` with an `openConnection( )` method that knows how to return a new instance of your subclass of `URLConnection`. Also provide a `getDefaultPort( )` method that returns the well-known port for the protocol. If your URL is not hierarchical, override `parseURL( )` and `toExternalForm( )` as well.
5. Implement the `URLStreamHandlerFactory` interface and the `createStreamHandler( )` method in a convenient class.

Let's look at handlers for two more protocols, daytime and chargen, which will bring up different challenges.

16.4.1. A daytime Protocol Handler

For a daytime protocol handler, let's say that the URL should look like *daytime:///vision.poly.edu*. We'll allow for nonstandard port assignments in the same way as with HTTP: follow the hostname with a colon and the port (*daytime:///vision.poly.edu:2082*). Finally, allow a terminating slash and ignore everything following the slash. For example, *daytime:///vision.poly.edu/index.html* is equivalent to *daytime:///vision.poly.edu*. This is similar enough to an *http* URL that the default `toExternalForm( )` and `parseURL()` methods will work.

Although the content returned by the daytime protocol is really text/plain, this protocol handler is going to reformat the data into an HTML page. Then it can return a content type of text/html and let the web browser display it more dramatically. The resulting HTML looks like this:

```
<html><head><title>The Time at metalab.unc.edu</title></head><body>
<h1>Fri Oct 29 14:32:07 1999</h1>
</body></html>
```

The trick is that the page can be broken up into three different strings:

- Everything before the time
- The time
- Everything after the time

The first and the third strings can be calculated before the connection is even opened. We'll formulate these as byte arrays of ASCII text and use them to create two `ByteArrayInputStream`s. Then we'll use a `SequenceInputStream` to combine those two streams with the data actually returned from the server. Example 16-4 demonstrates. This is a neat trick for protocols such as daytime that return a very limited amount of data; it can be inserted in a single place in an HTML document. Protocols such as finger that return more complex and less predictable text might need to use a `FilterInputStream` that inserts the HTML on the fly instead. And of course, a third possibility is to simply return a custom content type and use a custom content handler to display it. This third option is explored in the next chapter.

Example 16-4. The DaytimeURLConnection class

```
package com.macfaq.net.www.protocol.daytime;
```

```
import java.net.*;
import java.io.*;

public class DaytimeURLConnection extends URLConnection {

  private Socket connection = null;
  public final static int DEFAULT_PORT = 13;

  public DaytimeURLConnection (URL u) {
    super(u);
  }

  public synchronized InputStream getInputStream( ) throws IOException {

    if (!connected)  connect( );

    String header = "<html><head><title>The Time at "
     + url.getHost( ) + "</title></head><body><h1>";
    String footer = "</h1></body></html>";
    InputStream in1 = new ByteArrayInputStream(header.getBytes("8859_1"));
    InputStream in2 = this.connection.getInputStream( );
    InputStream in3 = new ByteArrayInputStream(footer.getBytes("8859_1"));

    SequenceInputStream result = new SequenceInputStream(in1, in2);
    result = new SequenceInputStream(result, in3);
    return result;

  }

  public String getContentType( ) {
    return "text/html";
  }

  public synchronized void connect( ) throws IOException {

    if (!connected) {
      int port = url.getPort( );
      if ( port <= 0 || port > 65535) {
        port = DEFAULT_PORT;
      }
      this.connection = new Socket(url.getHost( ), port);
      this.connected = true;
    }
  }
}
```

This class declares two fields. The first is `connection`, which is a `Socket` between the client and the server. The second field is `DEFAULT_PORT`, a `final static int` variable that holds the default port for the daytime protocol (port 13) and is used if the URL doesn't specify the port explicitly.

The constructor has no surprises. It just calls the superclass's constructor with the same argument, the `URL u`. The `connect()` method opens a connection to the specified server on the specified port (or, if no port is specified, to the default port); if the connection opens successfully, `connect( )` sets the `boolean` variable `connected` to `true`. Recall from the previous chapter that connected is a protected field in URLConnection that is inherited by this subclass. The Socket that's opened by this method is stored in the connection field for later use by getInputStream( ).

The getContentType( ) method returns a String containing a MIME type for the data. This method is called by the getContent( ) method of URLConnection to select the appropriate content handler. The getInputStream( ) method reformats the text into HTML, so the getContentType( ) method returns text/html.

The getInputStream( ) method builds a SequenceInputStream out of several string literals, the host property of url, and the actual stream provided by the Socket connecting the client to the server. If the socket is not connected when this method is called, the method calls connect( ) to establish the connection.

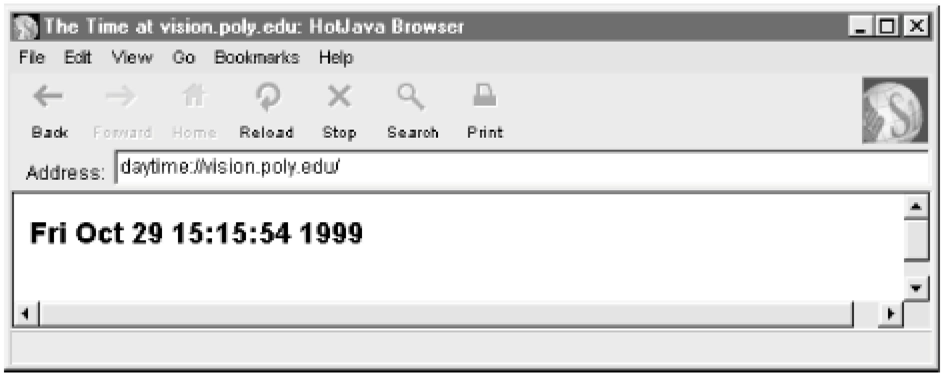
Next, you need a subclass of URLStreamHandler that knows how to handle a daytime server. This class needs an openConnection( ) method that builds a new DaytimeURLConnection from a URL and a getDefaultPort( ) method that returns the well-known daytime port 13. Since the daytime URL has been made similar to an *http* URL, we don't need to override parseURL(); once we have written openConnection ( ), we're done. Example 16-5 shows the daytime protocol's URLStreamHandler.

Example 16-5. The DaytimeURLStreamHandler class

```
package com.macfaq.net.www.protocol.daytime;

import java.net.*;
import java.io.*;

public class Handler extends URLStreamHandler {

  public int getDefaultPort( ) {
    return 13;
  }

  protected URLConnection openConnection(URL u) throws IOException {
    return new DaytimeURLConnection(u);
  }
}
```

Since we've used the same package-naming convention here as for the previous finger protocol handler, no further changes to HotJava's properties need to be made to let HotJava find this. Just compile the files, put the classes somewhere in HotJava's class path, and load a URL that points to an active daytime server. Figure 16-2 demonstrates.

Figure 16-2. HotJava using the daytime protocol handler


The Time at vision.poly.edu: HotJava Browser
File Edit View Go Bookmarks Help
Back Forward Home Reload Stop Search Print
Address: daytime://vision.poly.edu/

Fri Oct 29 15:15:54 1999

16.4.2. A chargen Protocol Handler

The chargen protocol, defined in RFC 864, is a very simple protocol designed for testing clients. The server listens for connections on port 19. When a client connects, the server sends an endless stream of characters until the client disconnects. Any input from the client is ignored. The RFC does not specify which character sequence to send but recommends that the server use a recognizable pattern. One common pattern is rotating, 72-character carriage return/linefeed delimited lines of the 95 ASCII printing characters, like this:

```
!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefgh
"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghi
#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghij
$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijk
%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijkl
&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklm
'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmn
()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmno
```

The big trick with this protocol is deciding when to stop. A TCP chargen server sends an unlimited amount of data. Most web browsers don't deal well with this. HotJava won't even attempt to display a file until it sees the end of the stream. Consequently, the first thing we'll need is a `FilterInputStream` subclass that cuts off the server (or at least starts ignoring it) after a certain amount of data has been sent. Example 16-6 is such a class.

Example 16-6. FiniteInputStream

```
package com.macfaq.io;

import java.io.*;
```

```
    public class FiniteInputStream extends FilterInputStream {

      private int limit = 8192;
      private int bytesRead = 0;

      public FiniteInputStream(InputStream in) {
        this(in, 8192);
      }

      public FiniteInputStream(InputStream in, int limit) {
        super(in);
        this.limit = limit;
      }

      public int read( ) throws IOException {

        if (bytesRead >= limit) return -1;
        int c = in.read( );
        bytesRead++;
        return c;

      }

      public int read(byte[] data) throws IOException {
        return this.read(data, 0, data.length);
      }

      public int read(byte[] data, int offset, int length)
       throws IOException {

            if (data == null) throw new NullPointerException( );
            else if ((offset < 0) || (offset > data.length) || (length < 0) ||
             ((offset + length) > data.length) || ((offset + length) < 0)) {
              throw new IndexOutOfBoundsException( );
            }
            else if (length == 0) {
              return 0;
            }

        if (bytesRead >= limit) return -1;
        else if (bytesRead + length > limit) {
          int numToRead = bytesRead + length - limit;
          int numRead = in.read(data, offset, numToRead);
          if (numRead == -1) return -1;
          bytesRead += numRead;
          return numRead;
        }
        else { // will not exceed limit
          int numRead = in.read(data, offset, length);
          if (numRead == -1) return -1;
          bytesRead += numRead;
          return numRead;
        }
      }

      public int available( ) throws IOException {
        if (bytesRead >= limit) return 1;
        else return in.available( );
    }
  }
```

Next, since there's no standard for the format of a chargen URL, we have to create one. Ideally, this should look as much like an *http* URL as possible. Therefore, we will implement a chargen URL like this:

```
chargen://hostname:port
```

Second, we need to choose the content type to be returned by the chargen protocol handler's `getContentType()` method. A chargen server returns ASCII text, so the `getContentType( )` method should return the string `text/plain`. The advantage of the `text/plain` MIME type is that Java already understands it.

Example 16-7 is a `ChargenURLConnection` class that subclasses `URLConnection`. This class overrides the `getContentType( )` and `getInputStream()` methods of `URLConnection` and implements `connect( )`. It also has a constructor that builds a new `URLConnection` from a URL.

Example 16-7. The ChargenURLConnection class

```
package com.macfaq.net.www.protocol.chargen;

import java.net.*;
import java.io.*;
import com.macfaq.io.*;

public class ChargenURLConnection extends URLConnection {

  private Socket connection = null;

  public final static int DEFAULT_PORT = 19;

  public ChargenURLConnection(URL u) {
    super(u);
  }

  public synchronized InputStream getInputStream( ) throws IOException {

    if (!connected) this.connect( );
    return new FiniteInputStream(this.connection.getInputStream( ));

  }

  public String getContentType( ) {
    return "text/plain";
  }

  public synchronized void connect( ) throws IOException {

    if (!connected) {
      int port = url.getPort( );
      if ( port < 1 || port > 65535) {
        port = DEFAULT_PORT;
      }
      this.connection = new Socket(url.getHost( ), port);
      this.connected = true;
    }
  }
}
```

This class has two fields. `connection` is a `Socket` between the client and the server. The second field is `DEFAULT_PORT`, a `final static int` that contains the chargen protocol's default port; this port is used if the URL does not specify the port explicitly.

The class's constructor just passes the `URL u` to the superclass's constructor. The `connect ( )` method opens a connection to the specified server on the specified port (or, if no port is specified, to the default chargen port, 19) and, assuming the connection is successfully opened, sets the `boolean` field `connected` to `true`. The `Socket` that `connect ( )` opens is stored in the field `connection` for later use by `getInputStream ( )`. The `connect ()` method is synchronized to avoid a possible race condition on the `connected` variable.

The `getContentType ( )` method returns a `String` containing a MIME type for the data. The data returned by a chargen server is always ASCII text, so this `getContentType ( )` method always returns `text/plain`.

The `getInputStream ( )` connects if necessary, then gets the `InputStream` from `this.connection`. Rather than returning it immediately, `getInputStream ( )` first chains it to a `FiniteInputStream`.

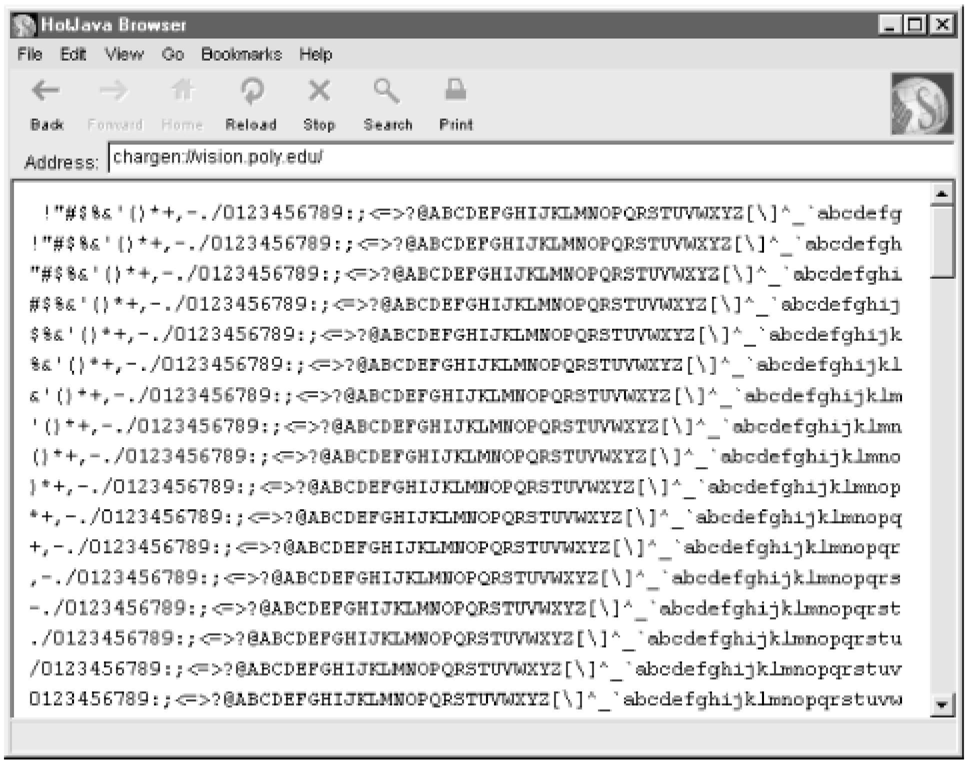
Now that we have a `URLConnection`, we need a subclass of `URLStreamHandler` that knows how to handle a chargen server. This class needs an `openConnection()` method that builds a new `ChargenURLConnection` from a URL and a `getDefaultPort ( )` method that returns the well-known chargen port. Since we defined the *chargen* URL so that it is similar to an *http* URL, we don't need to implement a `parseURL ( )` method. Example 16-8 is a stream handler for the chargen protocol.

Example 16-8. The chargen Handler class

```
package com.macfaq.net.www.protocol.chargen;

import java.net.*;
import java.io.*;

public class Handler extends URLStreamHandler {

  public int getDefaultPort( ) {
    return 19;
  }

  protected URLConnection openConnection(URL u) throws IOException {
    return new ChargenURLConnection(u);
  }
}
```

You can use HotJava to test this protocol handler. Run it and ask for a URL of a site running a chargen server, such as *vision.poly.edu*. Figure 16-3 shows the result.

Figure 16-3. HotJava using the chargen protocol handler


HotJava Browser
File Edit View Go Bookmarks Help
Back Forward Home Reload Stop Search Print
Address: chargen://vision.poly.edu/

```
 !"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefg
!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefgh
"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghi
#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghij
$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijk
%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijkl
&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklm
'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmn
()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmno
)*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnop
*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopq
+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqr
,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrs
-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrst
./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstu
/0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuv
0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvw
```

16.5. The URLStreamHandlerFactory Interface

The last section showed you how to install new protocol handlers that you wrote into HotJava, an application that someone else wrote. However, if you write your own application, you can implement your own scheme for finding and loading protocol handlers. The easiest way is to install a `URLStreamHandlerFactory` in the application:

```
public abstract interface URLStreamHandlerFactory
```

Only applications are allowed to install a new `URLStreamHandlerFactory`. Applets that run in the applet viewer or a web browser must use the `URLStreamHandlerFactory` that is provided. An attempt to set a different one will fail, either because another factory is already installed or because of a `SecurityException`.

The `URLStreamHandlerFactory` interface declares a single method, `createURLStreamHandler( )`:

```
public abstract URLStreamHandler createURLStreamHandler(String protocol)
```

This method loads the appropriate protocol handler for the specified protocol. To use this method, write a class that implements the `URLStreamHandlerFactory` interface and include a `createURLStreamHandler( )` method in that class. The method needs to know how to find the protocol handler for a given protocol. This step is no more complicated than knowing the names and packages of the custom protocols you've implemented.

The `createURLStreamHandler( )` method does not need to know the names of all the installed protocol handlers. If it doesn't recognize a protocol, it should simply return `null`, which tells Java to follow the default procedure for locating stream handlers; that is, to look for a class named *`protocol`*`.Handler` in one of the packages listed in the `java.protocol.handler.pkgs` system property or in the `sun.net.www.protocol` package.

To install the stream handler factory, pass an instance of the class that implements the `URLStreamHandlerFactory` interface to the static method `URL.setURLStreamHandlerFactory( )` at the start of the program. Example 16-9 is a `URLStreamHandlerFactory( )` with a `createURLStreamHandler( )` method that recognizes the finger, daytime, and chargen protocols and returns the appropriate handler from the last several examples. Since these classes are all named `Handler`, fully package-qualified names are used.

Example 16-9. A URLStreamHandlerFactory for finger, daytime, and chargen

```
package com.macfaq.net.www.protocol;

import java.net.*;

public class NewFactory implements URLStreamHandlerFactory {

   public URLStreamHandler createURLStreamHandler(String protocol) {

    if (protocol.equalsIgnoreCase("finger")) {
      return new com.macfaq.net.www.protocol.finger.Handler( );
    }
    else if (protocol.equalsIgnoreCase("chargen")) {
      return new com.macfaq.net.www.protocol.chargen.Handler( );
    }
    else if (protocol.equalsIgnoreCase("daytime")) {
      return new com.macfaq.net.www.protocol.daytime.Handler( );
    }
    else {
      return null;
    }
  }
}
```

Example 16-9 uses the `equalsIgnoreCase()` method from `java.lang.String` to test the identity of the protocol; it shouldn't make a difference whether you ask for *finger://rama.poly.edu* or *FINGER://RAMA.POLY.EDU*. If the protocol is recognized, `createURLStreamHandler( )` creates an instance of the proper `Handler` class and returns it; otherwise, the method returns `null`, which tells the `URL` class to look for a `URLStreamHandler` in the standard locations.

Since browsers, HotJava included, generally don't allow you to install your own `URLStreamHandlerFactory`, this will be of use only in applications. Example 16-10 is a simple character mode program that uses this factory and its associated protocol handlers to print server data on `System.out`. Notice that it does not import `com.macfaq.net.www.protocol.chargen`, `com.macfaq.net.www.protocol.finger`, or `com.macfaq.net.www.protocol.daytime`. All this program knows is that it has a URL. It does not need to know how that protocol is handled or even how the right `URLConnection` object is instantiated.

Example 16-10. A SourceViewer program that sets a URLStreamHandlerFactory

```
import java.net.*;
import java.io.*;
import com.macfaq.net.www.protocol.*;

public class SourceViewer4 {
  public static void main (String[] args) {

    URL.setURLStreamHandlerFactory(new NewFactory( ));

    if  (args.length > 0) {
      try {
        //Open the URL for reading
        URL u = new URL(args[0]);
        InputStream in = new BufferedInputStream(u.openStream( ));
        // chain the InputStream to a Reader
        Reader r = new InputStreamReader(in);
        int c;
        while ((c = r.read( )) != -1) {
          System.out.print((char) c);
        }
      }
      catch (MalformedURLException ex) {
        System.err.println(args[0] + " is not a parseable URL");
      }
      catch (IOException ex) {
        System.err.println(ex);
      }
    } //  end if
  } // end main
}  // end SourceViewer3
```

Aside from the one line that sets the `URLStreamHandlerFactory`, this is almost exactly like the earlier `SourceViewer` program in Example 7-5 (Chapter 7). For instance, here the program reads from a *finger* URL:

```
D:\JAVA\JNP2\examples\16>java SourceViewer4 finger://rama.poly.edu/
Login       Name               TTY         Idle    When    Where
nadats   Nabeel Datsun         pts/0         55 Fri 16:54  128.238.213.227
marcus   Marcus Tullius       *pts/1         20 Thu 12:12  128.238.10.177
marcus   Marcus Tullius       *pts/5       2:24 Thu 16:42  128.238.10.177
wri      Weber Research Insti  pts/10        55 Fri 13:26  rama.poly.edu
jbjovi   John B. Jovien        pts/9        25d Mon 14:54  128.238.213.229
```

Here it reads from a *daytime* URL:

```
% java SourceViewer4 daytime://tock.usno.navy.mil/
<html><head><title>The Time at tock.usno.navy.mil</title></head><body>
<h1>Fri Oct 29 21:22:49 1999
</h1></body></html>
```

However, it still works with all the usual protocol handlers that come bundled with the JDK. For instance here are the first few lines of output when it reads from an *http* URL:

```
% java SourceViewer4 http://www.oreilly.com/oreilly/about.html
<HTML>
<HEAD>
<TITLE>About O'Reilly &amp; Associates</TITLE>
</HEAD>
<BODY LINK="#770000" VLINK="#0000AA" BGCOLOR="#ffffff">

<table border=0 cellspacing=0 cellpadding=0 width=515>
<tr>
<td>
<img src="http://www.oreilly.com/graphics_new/generic_ora_header_wide.gif"
width="515" height="37" ALT="O'Reilly and Associates">
...
```