white paper

A DEVELOPER SERVICES COMPANY

developmentor™

Volume 3

A NEW ERA FOR JAVA PROTOCOL HANDLERS

Reliance on URLs in Java 2 Core APIs and Default Implementations Means Protocol Handlers Worth Knowing

Introduction

Up until Java 2, a real-world project had a lot of infrastructure to build: what with class loaders, security, distributed lookup and communications, etc. Between Java 1.1 and Java 2 Sun fleshed-out concrete implementations of what had been merely interesting abstractions:

- Security managers based on a concrete java.lang.SecurityManager implementation
- Secure class loading based on java.lang.SecureClassLoader
- A rich GUI framework provided by Swing
- And the motley crew of distributed application tools provided by Java 1.1 is a parade of frameworks in Java 2—it's difficult for a single programmer to even get a handle on the differences between Jini, EJB, JMX, FMA/Jiro, JNDI, JMS, etc.

Interestingly, of these powerful APIs and implementations, almost all have strong dependencies on URL objects. HTTP and FTP are good, general purpose protocols—definitely enough for general stream-based resource access. For fine-grained resource types like internationalisable resource strings, Java class file resources, and security policy resources, the workhorse HTTP and FTP protocols are long in the tooth and short on configurable performance. For faster, more efficient data flow you often need protocols designed for specific applications.

Based on a set of easily configurable factory patterns, the java.net.URL architecture allows you to implement easily a custom protocol handler and slip it into an application's java.net.URL class structure. This architecture has hardly changed since 1.0. Only now, with all the dependencies on URLs in Java 2's Core APIs and extensions, is it really worth your while to study this architecture and maybe even build your own custom protocol handlers. The really cool part is that your custom protocol handler is instantly usable by all of Java 2's advanced API implementations. For example, you can use a java.net.URLClassLoader to load classes using not just "file:"-, "http:"-, and "ftp:"-based classes and other resources, but also to access resources based on your own custom protocols—DB-based protocols, protocols with complex encryption and security, or maybe XML-based protocols (SOAP, XML-RPC, etc.).

In addition, custom protocols and URLs fit right into Java 2's configurable code-based security. A Java policy file that refers to your custom protocols will correctly sandbox code loaded using those protocols.

The myriad of systems coming from major vendors based on migrating code—such as Jini, JMX, FMA/Jiro, Java Embedded Server, etc.—are also based, ultimately, on the lowly java.net.URL class and its architecture of pluggable protocol handlers.

This paper describes for you everything you need to know to exploit the java.net.URL custom protocol plug-in architecture. After reading this paper, you'll have a good understanding of the java.net.URL architecture and how different java.net package objects interact to resolve URLs into resource streams. Knowing the architecture shows you how to build your own protocol handlers, which involves (at a minimum) providing a concrete implementation of two important java.net packet classes: java.net.URLStreamHandler and java.net.URLConnection. Once you've designed and created a custom protocol handler, you'll also need to know some tricks about deploying them in real-world Java execution environments. For most deployment scenarios, just placing a protocol handler on the classpath isn't enough. This paper details everything you need to know about deployment.

Once you have the architecture, know how to implement, and know how to deploy custom protocol handlers, you'll be able to integrate them into your Java-based applications with ease.

AN IMAGINARY REAL-WORLD EXAMPLE

I work on Java projects in the real world. And that almost always means the same thing: working from a version control system. Wouldn't it be great if I could refer to the class files and other resource files inside, say, a CVS (or some other version control system) server using **just a URL string**? I could run the latest version of a development project in progress without having to download all the latest files from the CVS system; without having to JAR them up or place them on you're the classpath; and without having to do all those painful, strange things Java developers do when working cooperatively on any but the smallest projects.

Imagine I have a VM that allows me to specify URLs in the classpath, not just files from the local file system[1]. If I had a CVS protocol handler as well, I could refer to my CVS project folders and preferred file version labels using URLs of this format:

```
cvs://myserver/myproject/bin#version_label
```

This URL would refer to the contents of the "bin" folder in the project named "myproject," on the server named "myserver." The presence of a "version_label" reference means that only the version of a file with that version label should be visible.

Now I'm busy developing an application, sharing code with my fellow developers through the version control system, in a project named "PagoPago." I want to run the latest binary code checked into the PagoPago project. I can do that without tedious downloads using this command line (again, imagining I've fashioned my VM to accept URLs as part of a classpath):

```
javavm –classpath cvs://codeserver/PagoPago/binaries \
        com.mycompany.KillerApp
```

The application automatically downloads all necessary code from the CVS server, it is not taken off my local file system. Suddenly, I notice a bug I don't think was there before. I can easily run the program again using the last version label of the application to see if the bug was there before:

```
javavm –classpath cvs://codeserver/PagoPago/binaries#V0.90 \
        com.mycompany.KillerApp
```

Java.net.URL's plug-in architecture allows you to define protocol handlers, such as the imaginary "cvs:" handler, and just start using them wherever you would use a URL. I can use the "cvs:" protocol handler as a class-loading codebase like I previously described. I could also use a "cvs:" URL in conjunction with a ResourceBundle to maintain different versions of my application's internationalized string resources.

The default SecurityManager in Java 2 uses code-based security to "sandbox" code based on where classes are loaded from and who digitally signed the classes. The policy files that describe the code-based security policy to the SecurityManager are URL-based. For example, if I wanted the classes loaded from my "cvs:" classpath to have unlimited access to system resources, all I have to do is add an appropriate grant into the current policy file:

```
grant codeBase "cvs://codeserver/PagoPago/bin/-"
{
  permission java.security.AllPermission;
};
```

Adding this grant into the current Java policy file would cause all code loaded from the CVS server to be trusted. Anywhere you use a URL in Java you can install and use a custom protocol handler.

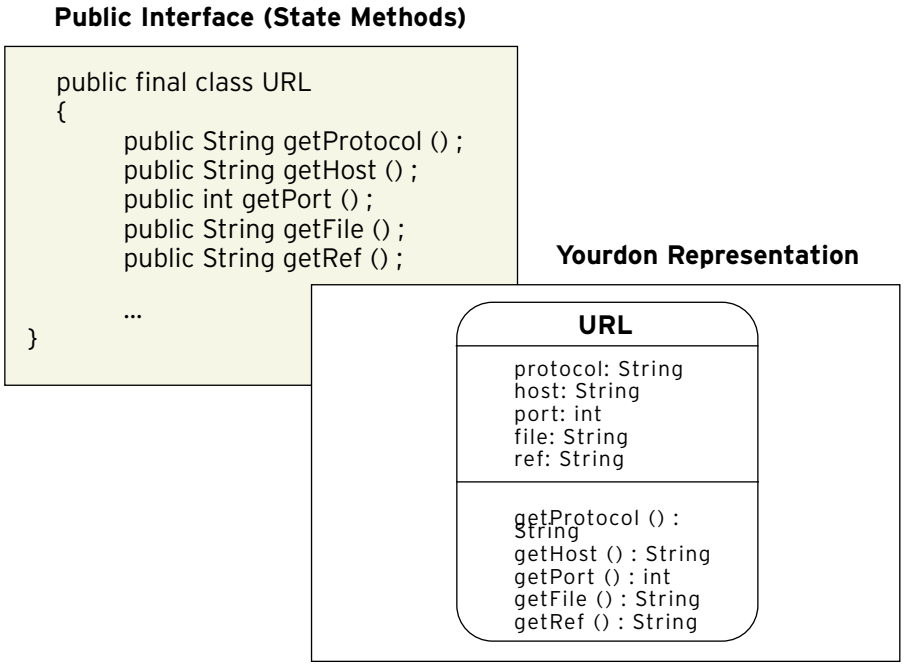
1 Actually, it's quite easy to fashion something like this yourself. Using Java 2's URLClassLoader and a "launcher" class you can make a network-enabled classpath rather easily.

java.net.URL Architecture

It was surprising to me to realize how minimal the java.net.URL class implementation really is. A java.net.URL object instance is used to represent a URL string, where a URL string **usually** follows this pattern:

```
protocol://host:port/filepath#ref
```

The public accessor methods of the URL class basically reflect the private fields of a URL object, and they also provide access to the individual parts of a URL string (See Figure 1).



• Figure 1: A URL object stores the contents of a URL string, parsed into several fields

Most of the action code in the java.net.URL class is dedicated to just the data representation in Figure 1: accessor method implementations, an equals() implementation to compare two URL objects for equivalency, a hashCode() implementation so URLs can easily be key objects in Maps, and a sameFile() method that compares two URL objects to see if everything save the "ref" field is the same.

RESOLVING A URL INTO A RESOURCE STREAM

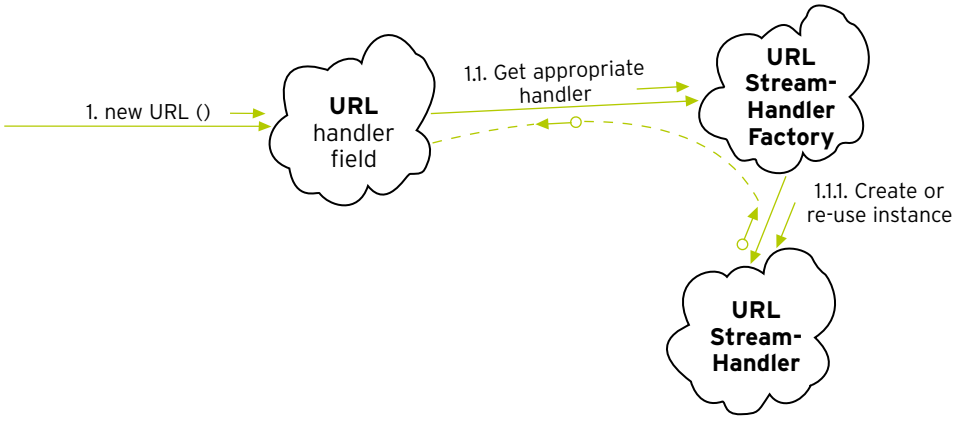
The URL class itself doesn't actually "know" how to access the resource stream the URL represents. Instead, when a URL object is asked to resolve itself into a resource stream—by calling the URL object's openStream() method—it delegates to a **protocol handler** object to get the job done. The implementation of a protocol handler object translates a URL object into a client/server connection, and ultimately into a resource stream. The URL object itself doesn't perform any network requests to a server. The protocol handler does that on behalf of the URL object.

• GETTING A PROTOCOL HANDLER OBJECT

There is a single, stateless protocol handler object for each unique protocol string ("http", "ftp", "file", "mailto", "cvs", etc.). A particular handler is delegated all responsibility for resource resolution of all URLs with the corresponding protocol string. That is, all "http:" URL objects are resolved using the HTTP protocol handler object, all "ftp:" URL objects are resolved by the FTP protocol handler, etc.

Physically, a protocol handler is a concrete subclass of the java.net.URLStreamHandler class. The URL object stores a reference to its corresponding URLStreamHandler internally during construction. The URL object uses a cached factory object known as the URLStreamHandlerFactory to obtain the reference to the appropriate URLStreamHandler. Figure 2 illustrates how a URL object gets an initial reference to its URLStreamHandler during construction.

Note that the previous description of a URLStreamHandlerFactory as a cached factory means the URLStreamHandlerFactory creates a URLStreamHandler instance the first time it is asked for a specific protocol. Thereafter it returns a reference to the same instance whenever asked for the same protocol. The factory uses a simple mechanism for resolving protocol name into URLStreamHandler-subclass class name, which I'll describe just a little bit later in this paper.



• Figure 2: Sequence of interactions between java.net objects to resolve a URL into a stream

One Cause of MalformedURLExceptions

Resolving a protocol string into a protocol handler happens at construction time. This explains why you can't create a URL object with an unknown protocol—in such cases the URL constructor throws a MalformedURLException. Take a look at the following example program:

```
public class BadProtoExample
{
  public static void main(String[] args)
  {
    try {
      URL url = new URL("bogus://www.microsoft.com");
      System.out.println("The URL: is: " + url);
```

```
    } catch (MalformedURLException mue) {
      System.err.println(mue);
    }
  }
}
```

This example program, when run on a typical Java VM, produces this output on the process' standard error stream:

```
java.net.MalformedURLException: unknown protocol: bogus
```

The Standard URLStreamHandlers

Sun's JRE defines several "standard" URLStreamHandlers for popular protocols: "http:", "ftp:", "mailto:", "gopher:" (?!), and even a special "jar:" protocol handler so you can access JAR file resources using URLs.

You can see these standard handlers, and associated implementation classes, in the JDK's RT.JAR file. Look for classes whose fully-qualified name starts with "sun.net.www.protocol". For example, the class "sun.net.www.protocol.http.Handler" defines the HTTP protocol handler. Class "sun.net.www.protocol.ftp.Handler" defines the FTP protocol handler class.

These are the standard URLStreamHandler implementations. Later in this paper I'll show you how to augment the list of handlers with your own, custom URLStreamHandler class implementations, and even how to override the standard implementations.

• OPENING THE STREAM

At some point after creating a URL object, controlling code will attempt to resolve that URL into a resource stream (that is, a java.io.InputStream). The simplest controlling code just calls the URL object's openStream() method and expects to get an InputStream instance returned.

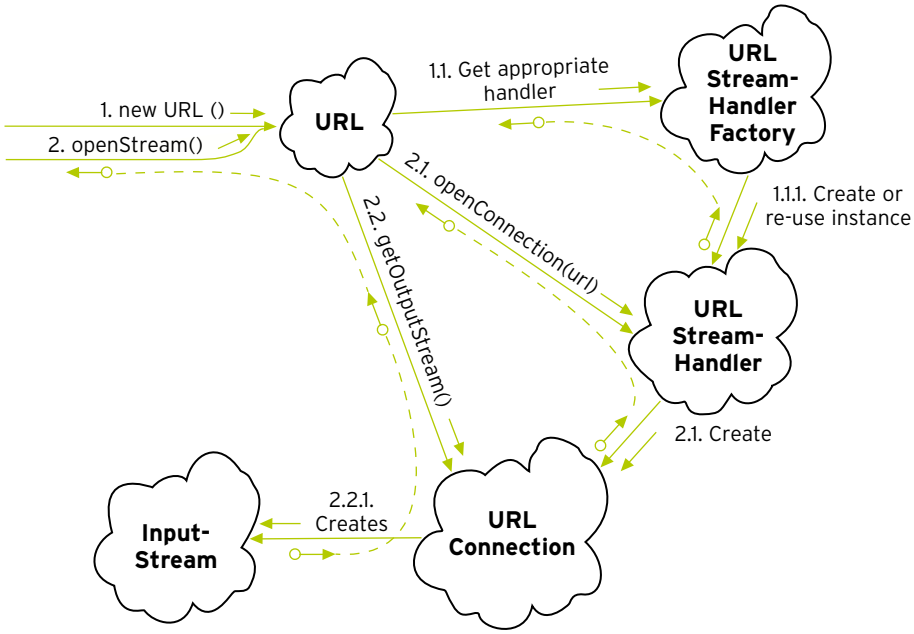
The URL object delegates all implementation of resource resolution to the URLStreamHandler and its helper classes and objects. The URLStreamHandler is stateless. An executing application may ask it to resolve several URLs into resource streams, possibly even simultaneously.

The URLStreamHandler actually creates a secondary object, known as the URLConnection, to perform an individual resource resolution. So, if a URLStreamHandler is asked to simultaneously resource 20 different URL objects into stream, all the URLStreamHandler does is create 20 different URLConnection objects, one to handle each of the 20 requests.

From an application's point of view, the URL object looks like it's doing a lot of work during the execution of this code:

```
URL url = new URL(someURLString);
InputStream resourceStream = url.openStream();
```

But in fact, the thread of execution spends very little time in actual URL class code. Instead, the URLStreamHandler and the URLConnection it creates accomplishes most of the resource resolution. The URLConnection object ultimately has the job of creating the OutputStream object. The URLConnection object is a task-oriented object. After creating the OutputStream object the URLConnection reaches the end of its lifecycle and can be destroyed. Figure 3 illustrates the flow of execution for the aforementioned two lines of sample code.

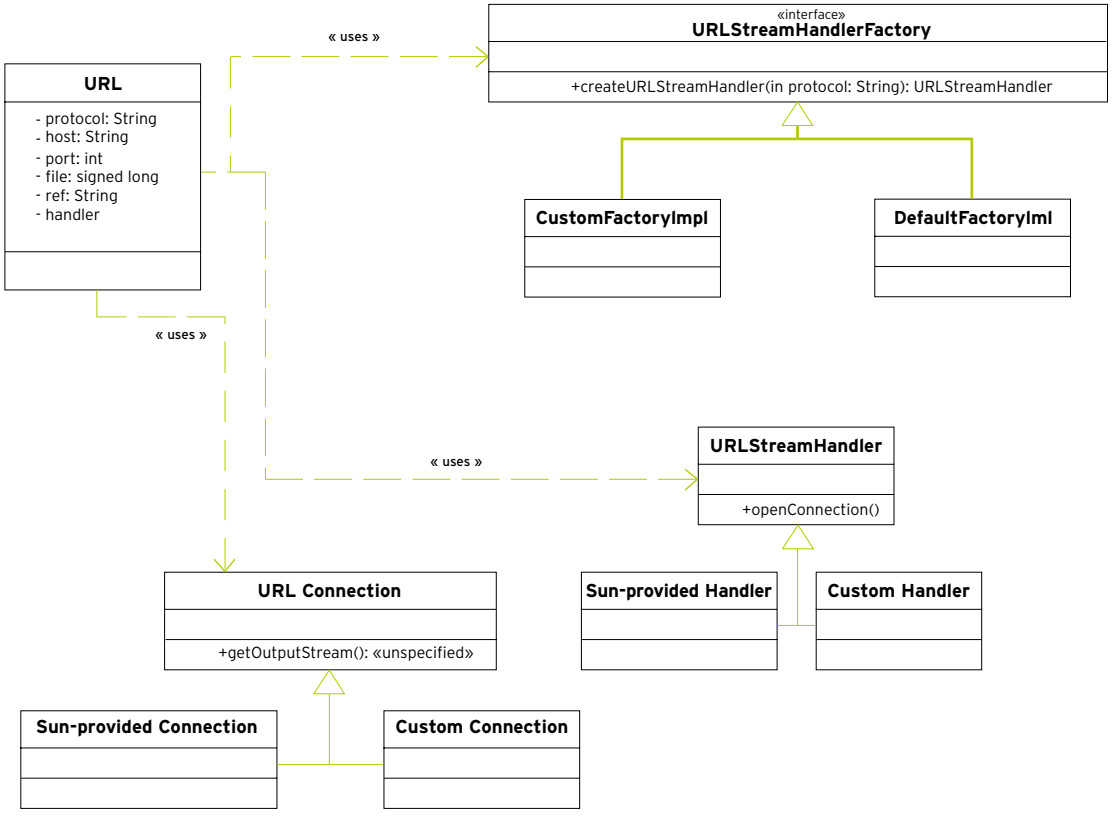


• Figure 3: Flow of execution while application creates then resolves a URL into an InputStream

Figure 3 shows the URL constructor using the URLStreamHandlerFactory to obtain a URLStreamHandler. When client code later calls openStream(), the URLStreamHandler is itself used as a factory to create a URLConnection. Most of the code that establishes a connection to a server and downloads a resource stream is located in the URLConnection. The URLConnection is a single-use object whose whole life's purpose is to generate an InputStream for the client code to consume.

CLASS RELATIONSHIPS

The URL class is a state storage class. The URL object stores information about the parts of a URL string, and manages other objects in order to resolve that URL string into a resource stream when asked to do so. Figure 4 shows the static relationship between classes in the java.net.URL architecture.



• Figure 4: Static relationship between classes and implementations in UML

The URL object uses a URLStreamHandlerFactory to create a URLStreamHandler. URLStreamHandlerFactory is in fact an interface. Java provides the default implementation of the factory interface, but you can replace it with your own—in certain deployment scenarios discussed later in this paper you **must** provide your own factory implementation.

The URL object maintains an immutable reference to its URLStreamHandler. When asked to resolve itself into a resource stream, the URL object uses its URLStreamHandler to create a URLConnection, then as part of the same code block the URL completes its use of the URLConnection. The URLConnection just represents a temporary, synchronous URLStreamHandler session. It's a place for the URLStreamHandler to place URL-object-specific state, which is necessary for stateless URLStreamHandlers.

The URLConnection is an interesting beast. Given the system as I've described it so far, the behavior of a URLConnection is only parameterizable by the URL object's state. That is, the only variables whose values differentiate the behavior of two URLConnection objects created by the same URLStreamHandler would be the field values of the URL objects (the host names, port numbers, filepaths, and ref strings stored internally by the URL objects) that create the two URLConnections. A request represented as a URL could not be augmented with any further information (such as HTTP POST data)—especially because the URL class is final, so subclasses can't add state members.

I'll describe URLConnection objects in more depth later. But I'll tell you now the URL class defines a public method openConnection() (as opposed to openStream()) whose implementation returns the URLConnection object itself back to controlling code, rather than an InputStream. This allows client code to parameterize further a resource request past what the state members of the URL object can represent. For example, a URLConnection created by an "http:" URL generates an HTTP POST request when its public interface is used in a particular way. The contents of a single "http:" URL string alone can't do that.

Implementing a Custom Protocol Handler

A minimal protocol handler requires a concrete URLStreamHandler class and a concrete URLConnection class. In addition, you may need to define several helper classes that assist the URLStreamHandler and URLConnection implementations, but the bare-bones implementation requires at least those two pieces.

I'm going to work through the creation of a protocol handler that accesses a local Windows registry. You can use a "win32registry" URL to access any Win32 registry key visible to the current user. The mapping between key names and URL strings is obvious, since the registry is just a persistent, hierarchical storage of binary, string, and integer data.

For anyone not familiar with a Windows registry, the aforementioned description pretty much states what a Windows registry is. It is a persistent tree, basically. Nodes are named with strings. Nodes can have values, which are strings, integers or arbitrary binary data. Nodes can also have sub-nodes. The Windows registry is used to store the configuration of a host machine: operating system parameters, devices drivers and start-up driver parameters, configuration of Windows services (equivalent to Unix daemons), and the configuration of third-party software applications. It's a huge storehouse of information about the local system.

And for any reader who is outwardly hostile to anything Microsoft-related, such as a Windows registry, I can only offer that the Windows registry's hierarchical arrangement is so close to the implicitly-hierarchical arrangement of resources implied by URL string syntax that a "win32registry:" protocol handler is a great example of the power of custom protocol handlers.

Normally you can only access a Windows registry in standard Java with native methods, because Microsoft only provides accessor functions in C-callable DLLs. With the "win32registry:" protocol handler, the protocol handler class implements all of the native code. Code that wants to access data in the registry need only reference the data with a "win32registry:" URL.

For example, if you wanted to know whether or not Sun's JRE was installed on the local system, you can easily find out by querying the Windows registry. JavaSoft adds registry entries storing the configuration of the JRE during JRE installation. Under the registry key, "HKEY_LOCAL_MACHINE\SOFTWARE\JavaSoft\Java Runtime Environment\1.2" is all the JRE configuration parameters, such as the installation directory of the JRE (stored in the "JavaHome" subkey).

Using the "win32registry:" protocol handler, you can look up the value of this key with just a few lines of code:

```
URL url = new URL("win32-registry:///HKEY_LOCAL_MACHINE/ \
        SOFTWARE/JavaSoft/Java Runtime Environment/1.2/JavaHome");
InputStream is = url.openStream();
DataInputStream dis = new DataInputStream(is);
String strJavaHome = dis.readUTF();
```

The code example shows that string-valued keys are encoded using UTF-encoded strings. Integer-valued keys will result in streams that include a 4-byte integer in network-byte order. Binary-valued keys will result in streams that just include the binary data of the key [2].

MAPPING URL STRINGS TO RESOURCE ADDRESSES

As with all good software projects, the first thing you have to do is planning. One of the main services a protocol handler performs is mapping a URL string into a physical resource location. Do this by defining a convention.

The "win32registry:" protocol handler maps a URL string to the local registry key. The key is always on a local machine [3]. The host name of the URL string must be empty or else the protocol handler will cause a MalformedURLException to be thrown when a URL object is created. That is, all valid URLs begin with

```
win32registry:///
```

After the third forward-slash, the "win32registry:" protocol handler interprets the rest of the URL string as just a registry key name (where forward-slashes are translated to the Windows-preferred blackslash). It ignores "#ref" strings.

Now that I've described how the "win32registry:" protocol handler will interpret and translate URL strings, I'm ready to start implementing it.

IMPLEMENTING URLSTREAMHANDLER

The first thing to implement is a concrete URLStreamHandler subclass. The URLStreamHandler implementation actually only has two relatively simple tasks to perform:

1. When a new "win32registry:" URL is created, the URLStreamHandler must parse the URL string into the various field values host, port, file, and ref.

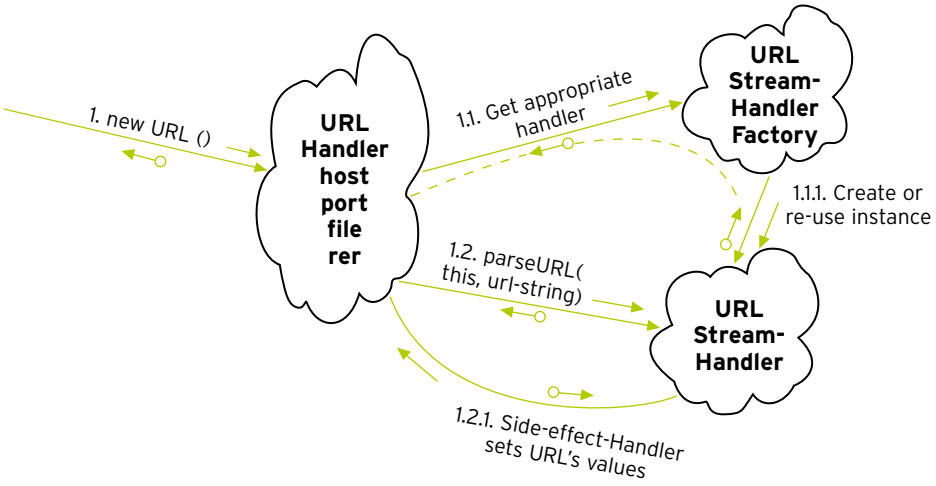
2. Create and return a new URLConnection instance when the URLStreamHandler's openConnection() method is called.

[2] Probably a more general and useful solution would have used XML documents instead of DataInputStream-readable streams. However, this is just an example protocol handler, I didn't want my description to get bogged down with ancillary APIs such as the org.xml APIs.

[3] The Windows registry API does allow an application to look at the registry on foreign machines within a domain. However, machines must be identified by their NetBios name. The URL class strictly requires that any host names provided in the URL string can be turned into a java.net.InetAddress object (using InetAddress.getByName()). Because you can't necessarily do that conversion with NetBios names, my "win32registry:" protocol handler simplifies things by restricting access to the local machine.

• PARSING URL STRINGS

Task 1 is actually quite important. Whenever a new URL object is created, the URL constructor first obtains a reference to the appropriate URLStreamHandler (using the URLStreamHandlerFactory, as previously described), and second asks the URLStreamHandler to parse the URL string into host, port, file and ref field values. Figure 5 illustrates this two-part construction process.



• Figure 5: Two-part construction of a URL object. The handler populates the URL's fields.

The URL constructor calls handler.parseURL(). Your implementation of parseURL() is supposed to parse the URL string and use the embedded values to populate the URL object's host, port, file, and ref fields.

Rather than every protocol handler designer re-writing the same code to parse a URL into its constituent parts, the default implementation of parseURL() assumes the URL is in the normal "**protocol://host:port/file#ref**" format. So, if your handlers expects URLs to be of that form, you don't actually have to implement parsing code [4]. The "win32registry:" protocol handler expects URLs to be of the normal form, so there's no need to override the inherited parseURL() method implementation.

• CREATING URLCONNECTION INSTANCES

The second thing a concrete URLStreamHandler must do is provide a concrete implementation of the openConnection() method. This is an abstract method declared in the URLStreamHandler base class:

```
public abstract class URLStreamHandler
{
  ...
  protected abstract URLConnection openConnection(URL u);
  ...
}
```

[4] Consider an alternative example: the "mailto:" protocol handler. "Mailto:" URLs don't look like normal URLs, so a "mailto:" protocol handler implementation must override the parseURL() method and provide an alternative parsing algorithm.

The handler's openConnection() method is called directly by the URL class openStream() implementation. An openConnection() call is a request to the handler that it create an object that can resolve the URL into a resource stream. The URLConnection object is that object.

The name, "URLConnection," indicates that the java.net.URL architecture developers expected the only way to resolve a URL object into a stream is actually to open a connection to some server, like an HTTP or an FTP server. My "win32registry:" handler actually doesn't have to open a network connection to any other server. Still, I must provide a URLConnection implementation that effects the same behavior – my connection implementation won't contact a different server but will instead contact the Windows operating system through a series of native calls.

A deep explanation of the "win32registry:" URLConnection implementation follows in a moment. To complete the "win32registry:" handler's openConnection() implementation, all I have to do is create and return my custom URLConnection instance:

```
public class Handler extends java.net.URLStreamHandler
{
  protected URLConnection openConnection(URL u)
  {
    //...create and return a custom URLConnection initialized
    //   with a reference to the target URL object...
    return new RegistryURLConnection(u);
  }
}
```

In fact, the aforementioned code is just about my entire URLStreamHandler implementation; I left most of the implementation for the RegistryURLConnection class.

• PACKAGE AND CLASS NAMING CONVENTION

One somewhat unconventional requirement of URLStreamHandler classes is that the class name and even the package name have certain restrictions. You must name the handler class Handler, as in the previous example.

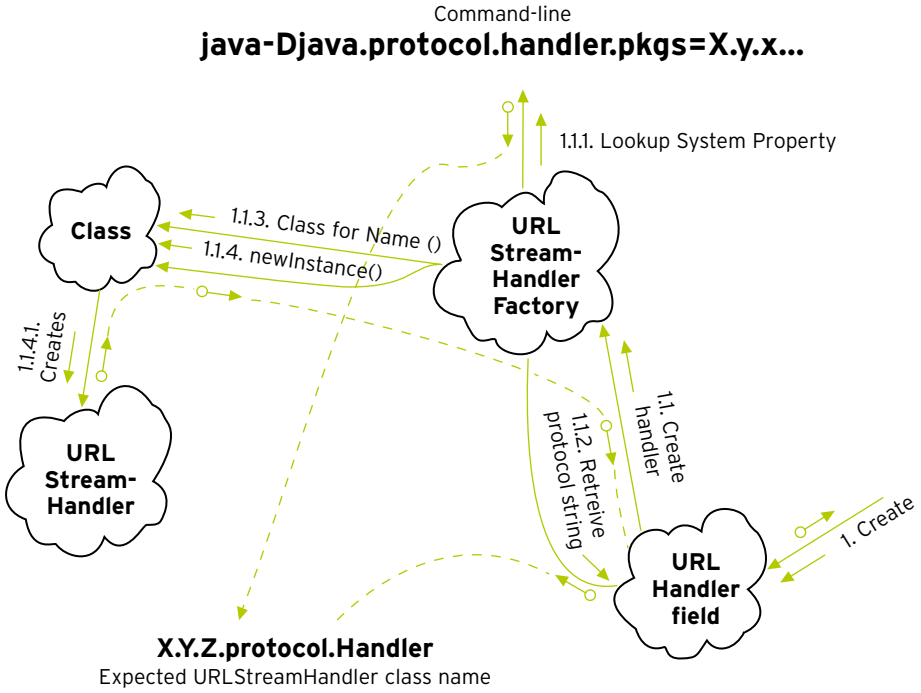
The package name must include the protocol name as the last dot-separated token. Any of these fully-qualified names are valid names for my "win32registry:" handler class:

- com.develop.protocols.win32-registry.Handler
- some.package.win32-registry.Handler
- some.other.package.win32-registry.Handler

That is, the fully-qualified class name must end in "win32registry.Handler".

The reason for this unorthodox naming restriction has to do with how the URLStreamHandlerFactory maps protocol names to handler classes. Remember that when a URL is created, the URL constructor code asks the URLStreamHandlerFactory for the URLStreamHandler associated with the URL's protocol. As it turns out, the factory just uses the protocol name, combined with a well-known system property, to generate a fully-qualified class name.

The "java.protocol.handler.pkgs" system property is a list of package name prefixes used by the URLStreamHandlerFactory to resolve protocol names into actual handler class names. For example, imagine this property has the value "com.develop.protocols". When asked to find a handler class for the "win32-registry:" protocol, the URLStreamHandlerFactory concatenates the system property value ("com.develop.protocols") with the protocol name ("win32-registry") and the expected class name "Handler". The resultant fully-qualified class name is "com.develop.protocols.win32-registry.Handler", and that's the class name the factory looks for on the classpath. Figure 6 illustrates the sequence controlled by the URLStreamHandlerFactory for creating the URLStreamHandler associated with a protocol string.



• Figure 6: URLStreamHandlerFactory creates the URLStreamHandler using reflection to first load appropriate class, then create instance.

The "java.protocol.handler.pkgs" system property may contain a list of package prefixes, rather than just a single-package prefix name. The list is pipe-character ('|') separated. The URLStreamHandlerFactory evaluates each prefix in right-to-left order, stopping when it finds the first generated fully-qualified class name matching a class on the classpath.

Note also that the URLStreamHandlerFactory always appends the default protocol handler package prefix, "sun.net.www.protocol", to this system property. That is, Sun's default handlers are always used if a handler can't be found in a different package first.

The "win32registry:" handler must be in an appropriately named package:

```
package com.develop.protocols.win32registry;
public class Handler extends java.net.URLStreamHandler
{
...
}
```

The VM must be started up with an appropriate value for the "java.protocol.handler.pkgs" system property:

```
java –Djava.protocol.handler.pkgs=com.develop.protocols ...
```

• IMPLEMENTING URLSTREAMHANDLERFACTORY

An alternative to using the "XYZ.protocol.Handler" package and class naming convention (along with the "java.protocol.handler.pkgs" system property) previously described is implementing your own URLStreamHandlerFactory. The default URLStreamHandlerFactory pays attention to the "java.protocol.handler.pkgs" system property and applies the package and class naming conventions. Your alternative implementation of the URLStreamHandlerFactory interface could get a handler implementation from wherever it wanted.

There are a few situations where you might consider using your own factory implementation. Foremost is when you don't have access to set the "java.protocol.handler.pkgs" system property. This property must be set at VM start-up time, and is usually set by a command-line parameter or a start-up properties file that the VM uses. If you want to use your custom protocol as part of a component, such as a Servlet, Jini service or EJBean implementation running within a container VM, then you will probably need to include a URLStreamHandlerFactory implementation as part of your component installation.

The URLStreamHandlerFactory interface is very simple:

```
public interface URLStreamHandlerFactory
{
  public URLStreamHandler createURLStreamHandler(String protocol);
}
```

Get a URL object to use your custom factory implementation instead of the default by using an overloaded version of the URL class constructor, as in this example:

```
URL url = new URL(null, "cvs://server/project/folder#version",
    new MyURLStreamHandlerFactoryImpl());
```

This URL object will use MyURLStreamHandlerFactoryImpl instead of the default factory. Note, however, that Java 2 security does require a NetPermission named "specifyStreamHandler" granted to the calling context in order to use this constructor. That is, code without that NetPermission grant, running within a Java 2 VM with a default SecurityManager will receive an AccessControlException. The following java.policy file grant demonstrates how to assign such a permission to code signed by "MyCompany":

```
...
grant signedBy "MyCompany" {
  permission java.net.NetPermission "specifyStreamHandler";
};
...
```

IMPLEMENTING URLCONNECTION

A URLConnection object manages the translation of a URL object into a resource stream. URLConnections in general can handle both interactive protocols, such as HTTP and FTP, as well as non-interactive protocols, such as "jar:" and the "win32registry:" protocol. That is, the URLConnection subclass used to make HTTP requests is able to handle interactive request/response dialogs with a server. The RegistryURLConnection implementation only needs to make certain local system calls to translate a URL object into a resource stream—there is no complex, multi-part request/response dialogue between the RegistryURLConnection and the native O/S.

The URLConnection class contract must be able to handle **both** types of resource request models. In addition, do so in a very generic fashion: the interface must be easily applied to many different types of client/server interactions. Consequently, the URLConnection class' design is somewhat abstract and rather complicated. I will explain URLConnections in general before describing my implementation of my RegistryURLConnection class in particular.

• URLCONNECTION INTERFACE AND CONTRACT WITH CONTROLLING CODE

A URLConnection is a collection of **request** and **response headers**, and optionally also an OutputStream (described by request headers) and an InputStream (described by response headers). The public API of the URLConnection class is broken down into methods that access or modify request headers, methods that access or modify response headers, and methods to access the InputStream and OutputStream.

```
public abstract class URLConnection
{
  //...methods to access and manipulate request properties...
  public void setRequestProperty(String name, String value);
  public String getRequestProperty(String name);

  //...methods to access response properties,
  //   known simply as "headers" in this API.
  public String getHeaderField(int n);
  public String getHeaderFieldKey(int n);
  public String getHeaderField(String name);

  //...Convenience implementations for commonly-used header
```

```
  //   fields. Implementations just call getHeaderField()...
  public int getContentLength();
  public String getContentEncoding();
  public String getContentType();

  ...
  //...get an OutputStream directly connected to server. Only
protocols
  //   that support writing data to a server implement this...
  public OutputStream getOutputStream();

  //...get an InputStream directly connected to server, contains
  //   resource stream sent from server. Only protocols supporting
  //   a stream-based response from the server implement this.
  public InputStream getInputStream();

  ...
}
```

All URLConnections must support the notion of request and response headers. The request headers describe the request being made, and the response headers describe the result of the request.

Some protocols may also support the client sending a stream of data as part of the request. For example, an HTTP POST request includes a body that's a stream of data. A protocol that does support interaction will have an implementation of getOutputStream(). Protocols that don't support interaction (like the "win32registry:" protocol) will simply throw an UnknownServiceException from getOutputStream().

Similarly, **some** protocols support a response stream. HTTP, FTP, and my "win32registry:" protocol all support the concept of a response stream. In fact, any protocol that can map a URL to a response stream obviously support this concept. A counter example is the "mailto:" protocol. A "mailto:" request does not include any kind of response stream—any information you need to gather about the success of a "mailto:" request can be gathered from the response headers alone. Controlling code accesses the response stream of a URLConnection request using the getInputSteam() method. If the protocol doesn't support response streams, getInputStream() throws an UnknownServiceException.

When first created, the field values of the associated URL object initialize the URLConnection object. The URLConnection object has not attempted to access any resource, and hasn't opened any connection to a server. Basically, it has undergone only **primary initialization**. This means the field values of the associated URL object may have initialized some of the request headers.

Controlling code with a reference to the URLConnection object may call setRequestHeader() to further parameterize the request.

So far in this paper, I've suggested that "controlling code" of a URLConnection instance is the openStream() implementation in the URL class. That is, "controling code" is the only block of code that can directly touch a URLConnection object. As it turns out, the URL.openConnection() method causes a new URLConnection object to be created (through the process already defined), but then simply hands the URLConnection back to the caller. This way, code that created a URL object may have direct access to its URLConnection, and that controlling code can then manipulate the request the URLConnection is to make. The following example code shows how to perform an HTTP PORT request in Java using a "http:" URL's URLConnection.

```
// Application (a.k.a "controlling") code.

URL url = new URL("http://someserver/file");
URLConnection urlconn = url.openConnection();

//...use setRequestHeader() to modify request into
//   a POST request. (The "METHOD" header is specially
//   understood by the HTTP URLConnection.)...
urlconn.setRequestHeader("METHOD", "POST");

//...Indicate the content type of the POST data...
urlconn.setContentType("application/x-www-form-urlencoded");
 //...equivalent to setting the "content-type" header...

//...Open a connection with the server...
urlconn.connect();

//...Get the output stream to send request data...
OutputStream os = urlconn.getOutputStream();

//...Write POSTed form data...

//...Get the InputStream, which completes the request...
InputStream response = urlconn.getInputStream();
```

Once the controlling code has initialized all necessary request headers, the controlling code calls the URLConnection's connect() method. This method has a nebulous definition in general. Basically, it means that the request headers have been fully-formed, and for the URLConnection to go ahead and make a connection to the server/service/library that is servicing the request. An HTTP URLConnection implementation would obviously make a TCP connection and send request headers in response to this method being called. The "win32-registry:" protocol handler goes through native methods to actually get the target registry key value in response to this method. Note that getOutputStream() and getInputStream() methods both call connect() initially if it hasn't been called yet.

When connect() returns, the response headers should be initialized to values indicating the result of the request.

Controlling code expecting a response stream would call getInputStream() at this point. The contents of the stream and the response header values indicate the result of the URL request.

Again, the URLConnection base class has a rather generic definition, meant to be a broad umbrella under which the concepts of parameterized requests and responses in general can fit. When implementing a protocol handler you must understand how your protocol fits into this generic model so that you can implement a meaningful URLConnection class. I recommend you read the JavaDocs for the java.net.URLConnection class. There are several convenience and hook methods in the API I left out or glossed over in order to discuss the essential points.

• THE REGISTRYURLCONNECTION IMPLEMENTATION

After that lengthy description of the URLConnection class, I think you'll be surprised to see how simple the RegistryURLConnection implementation looks. After all, my "win32registry:" protocol doesn't support an interactive

OutputStream, and it can ignore any request header settings made by controlling code. That is, because all information needed to find a resource is located in the URL object's fields, there's no need for my implementation to pay attention to further parameterization by controlling code. The setRequestHeader() implementation is empty:

```
package com.develop.protocols;

public class RegistryURLConnection extends java.net.URLConnection
{

  ...
  public void setRequestHeader(String name, String value) { }
  ...
}
```

Meaning of RegistryURLConnection's Response Headers

My implementation does have to do a little cognitive mapping to squeeze a registry value response into the URLConnection's response headers and OutputStream. First of all, RegistryURLConnection.connect() sets a single response header called "Exists" to indicate whether or not the target registry key exists. The valid values will be "True" or "False".

Next, if a registry key does exist, that doesn't mean it has a value. For example, the \HKEY_LOCAL_MACHINE\SOFTWARE\JavaSoft\Java Runtime Environment\1.2 key doesn't have a value, it's just a parent node to sub-nodes that actually do have values. A request for this node would result in a value of "True" for the "Exists" response header, but a value of "application/x-null" for the "Content-type" response header.

The other valid "Content-type" header field value, "application/x-java-DataStream," is used if the key does exist and it does have a value associated with it. In this case, another header field, "DataType" indicates the registry key's value. This response header will have one of three values: "UTF", "int", or "binary".

In the case of "binary", the "Content-length" header field indicates the length of the response stream.

The implementation of all the getXYZ() methods to access response headers forward the call to the getHeaderField() method. The getHeaderField() implementation just accesses values in a Map object. This is a minimal implementation of response headers that supports actually having response headers (as opposed to a null implementation, used by protocol handlers that don't use response headers at all).

Where to Find the "win32registry:" Protocol Handler Implementation

The full source code, including examples you can run to test the protocol handler, is available by sending e-mail to: javaprotocol@develop.com.

Summary

The Java 2 Core API depends heavily on the URL class. A deep understanding of the internals of the java.net.URL architecture allows you to exploit those dependencies by creating reusable, plug-in protocol handlers. Once you build a protocol handler you can use it in any Java VM, creating URL objects that refer to your new protocol.

It takes very little work at deployment time to enable your handler: just let the VM know about your handler by defining the "java.protocol.handler.pkgs" system property, and make sure your handler implementation is available off the classpath.

Using a custom protocol handler can often save you vast amounts of time and effort when designing and implementing Java architectures to access stream-based resources. The java.net.URL architecture is already built, already a part of all Java 1.x+ installations, and used by many parts of the Java Core API, Extension APIs, and third-party packages. There's no reason not to exploit this architecture whenever possible.