ORACLE PRESS

CORE JAVA

Volume II: Advanced Features

TWELFTH EDITION

ORACLE

Cay S. Horstmann

Core Java

Volume II: Advanced Features

Twelfth Edition

java.net.InetSocketAddress 1.4

- `InetSocketAddress(String hostname, int port)`

  constructs an address object with the given host and port, resolving the host name during construction. If the host name cannot be resolved, the address object's `unresolved` property is set to `true`.

- `boolean isUnresolved()`

  returns `true` if this address object could not be resolved.

java.nio.channels.SocketChannel 1.4

- `static SocketChannel open(SocketAddress address)`

  opens a socket channel and connects it to a remote address.

java.nio.channels.Channels 1.4

- `static InputStream newInputStream(ReadableByteChannel channel)`

  constructs an input stream that reads from the given channel.

- `static OutputStream newOutputStream(WritableByteChannel channel)`

  constructs an output stream that writes to the given channel.

4.3 Getting Web Data

To access web servers in a Java program, you will want to work at a higher level than socket connections and HTTP requests. In the following sections, we discuss the classes that the Java library provides for this purpose.

4.3.1 URLs and URIs

The `URL` and `URLConnection` classes encapsulate much of the complexity of retrieving information from a remote site. You can construct a `URL` object from a string:

```
var url = new URL(urlString);
```

If you simply want to fetch the contents of the resource, use the `openStream` method of the `URL` class. This method yields an `InputStream` object. Use it in the usual way—for example, to construct a `Scanner`:

```
InputStream inStream = url.openStream();
var in = new Scanner(inStream, StandardCharsets.UTF_8);
```

The `java.net` package makes a useful distinction between URLs (uniform resource *locators*) and URIs (uniform resource *identifiers*).

A URI is a purely syntactical construct that contains the various parts of the string specifying a web resource. A URL is a special kind of URI, namely one with sufficient information to *locate* a resource. Other URIs, such as

```
mailto:cay@horstmann.com
```

are not locators—there is no data to locate from this identifier. Such a URI is called a URN (uniform resource *name*).

In the Java library, the `URI` class has no methods for accessing the resource that the identifier specifies—its sole purpose is parsing. In contrast, the `URL` class can open a stream to the resource. For that reason, the `URL` class only works with schemes that the Java library knows how to handle, such as `http:`, `https:`, `ftp:`, the local file system (`file:`), and JAR files (`jar:`).

To see why parsing is not trivial, consider how complex URIs can be. For example,

```
http:/google.com?q=Beach+Chalet
ftp://username:password@ftp.yourserver.com/pub/file.txt
```

The URI specification gives the rules for the makeup of these identifiers. A URI has the syntax

[*scheme*:]*schemeSpecificPart*[#*fragment*]

Here, the `[. . .]` denotes an optional part, and the `:` and `#` are included literally in the identifier.

If the *scheme*: part is present, the URI is called *absolute*. Otherwise, it is called *relative*.

An absolute URI is *opaque* if the *schemeSpecificPart* does not begin with a `/` such as

```
mailto:cay@horstmann.com
```

All absolute nonopaque URIs and all relative URIs are *hierarchical*. Examples are

```
http://horstmann.com/index.html
../../java/net/Socket.html#Socket()
```

The *schemeSpecificPart* of a hierarchical URI has the structure

[//*authority*][*path*][?*query*]

where, again, `[. . .]` denotes optional parts.

For server-based URIs, the *authority* part has the form

[*user-info*@]*host*[:*port*]

The *port* must be an integer.

RFC 2396, which standardizes URIs, also supports a registry-based mechanism in which the *authority* has a different format, but this is not in common use.

One of the purposes of the `URI` class is to parse an identifier and break it up into its components. You can retrieve them with the methods

```
getScheme
getSchemeSpecificPart
getAuthority
getUserInfo
getHost
getPort
getPath
getQuery
getFragment
```

The other purpose of the `URI` class is the handling of absolute and relative identifiers. If you have an absolute URI such as

```
http://docs.mycompany.com/api/java/net/ServerSocket.html
```

and a relative URI such as

```
../../java/net/Socket.html#Socket()
```

then you can combine the two into an absolute URI.

```
http://docs.mycompany.com/api/java/net/Socket.html#Socket()
```

This process is called *resolving* a relative URL.

The opposite process is called *relativization*. For example, suppose you have a *base* URI

```
http://docs.mycompany.com/api
```

and a URI

```
http://docs.mycompany.com/api/java/lang/String.html
```

Then the relativized URI is

```
java/lang/String.html
```

The URI class supports both of these operations:

```
relative = base.relativize(combined);
combined = base.resolve(relative);
```

4.3.2 Using a URLConnection to Retrieve Information

If you want additional information about a web resource, you should use the `URLConnection` class, which gives you much more control than the basic `URL` class.

When working with a `URLConnection` object, you must carefully schedule your steps.

1. Call the `openConnect ion` method of the `URL` class to obtain the `URLConnection` object:

   ```
   URLConnection connection = url.openConnection();
   ```

2. Set any request properties, using the methods

   ```
   setDoInput
   setDoOutput
   setIfModifiedSince
   setUseCaches
   setAllowUserInteraction
   setRequestProperty
   setConnectTimeout
   setReadTimeout
   ```

   We discuss these methods later in this section and in the API notes.

3. Connect to the remote resource by calling the `connect` method:

   ```
   connection.connect();
   ```

   Besides making a socket connection to the server, this method also queries the server for *header information.*

4. After connecting to the server, you can query the header information. Two methods, `getHeaderFieldKey` and `getHeaderField`, enumerate all fields of the header. The method `getHeaderFields` gets a standard `Map` object containing the header fields. For your convenience, the following methods query standard fields:

   ```
   getContentType
   getContentLength
   getContentEncoding
   getDate
   getExpiration
   getLastModified
   ```

5. Finally, you can access the resource data. Use the `getInputStream` method to obtain an input stream for reading the information. (This is the same input stream that the `openStream` method of the `URL` class returns.) The other method, `getContent`, isn't very useful in practice. The objects that are returned by standard content types such as `text/plain` and `image/gif` require

classes in the com.sun hierarchy for processing. You could register your own content handlers, but we do not discuss this technique in our book.

CAUTION: Some programmers form a wrong mental image when using the URLConnection class, thinking that the getInputStream and getOutputStream methods are similar to those of the Socket class. But that isn't quite true. The URLConnection class does quite a bit of magic behind the scenes—in particular, the handling of request and response headers. For that reason, it is important that you follow the setup steps for the connection.

Let us now look at some of the URLConnection methods in detail. Several methods set properties of the connection before connecting to the server. The most important ones are setDoInput and setDoOutput. By default, the connection yields an input stream for reading from the server but no output stream for writing. If you want an output stream (for example, for posting data to a web server), you need to call

```
connection.setDoOutput(true);
```

Next, you may want to set some of the request headers. The request headers are sent together with the request command to the server. Here is an example:

```
GET www.server.com/index.html HTTP/1.0
Referer: http://www.somewhere.com/links.html
Proxy-Connection: Keep-Alive
User-Agent: Mozilla/5.0 (X11; U; Linux i686; en-US; rv:1.8.1.4)
Host: www.server.com
Accept: text/html, image/gif, image/jpeg, image/png, */*
Accept-Language: en
Accept-Charset: iso-8859-1,*,utf-8
Cookie: orangemilano=192218887821987
```

The setIfModifiedSince method tells the connection that you are only interested in data modified since a certain date.

Finally, you can use the catch-all setRequestProperty method to set any name/value pair that is meaningful for the particular protocol. For the format of the HTTP request headers, see RFC 2616. Some of these parameters are not well documented and are passed around by word of mouth from one programmer to the next. For example, if you want to access a password-protected web page, you must do the following:

1. Concatenate the user name, a colon, and the password.

   ```
   String input = username + ":" + password;
   ```

2. Compute the Base64 encoding of the resulting string. (The Base64 encoding encodes a sequence of bytes into a sequence of printable ASCII characters.)

   ```
   Base64.Encoder encoder = Base64.getEncoder();
   String encoding = encoder.encodeToString(input.getBytes(StandardCharsets.UTF_8));
   ```

3. Call the `setRequestProperty` method with a name of `"Authorization"` and the value `"Basic " + encoding`.

   ```
   connection.setRequestProperty("Authorization", "Basic " + encoding);
   ```

✓ **TIP:** You just saw how to access a password-protected web page. To access a password-protected file by FTP, use an entirely different method: Construct a URL of the form

```
ftp://username:password@ftp.yourserver.com/pub/file.txt
```

Once you call the `connect` method, you can query the response header information. First, let's see how to enumerate all response header fields. The implementors of this class felt a need to express their individuality by introducing yet another iteration protocol. The call

```
String key = connection.getHeaderFieldKey(n);
```

gets the `n`th key from the response header, where `n` starts from `1`! It returns `null` if `n` is zero or greater than the total number of header fields. There is no method to return the number of fields; you simply keep calling `getHeaderFieldKey` until you get `null`. Similarly, the call

```
String value = connection.getHeaderField(n);
```

returns the `n`th value.

The method `getHeaderFields` returns a `Map` of response header fields.

```
Map<String,List<String>> headerFields = connection.getHeaderFields();
```

Here is a set of response header fields from a typical HTTP request:

```
Date: Wed, 27 Aug 2008 00:15:48 GMT
Server: Apache/2.2.2 (Unix)
Last-Modified: Sun, 22 Jun 2008 20:53:38 GMT
Accept-Ranges: bytes
Content-Length: 4813
Connection: close
Content-Type: text/html
```