Table of Contents

Chapter 17. Content Handlers... 1
17.1. What Is a Content Handler?.. 3
17.2. The ContentHandler Class.. 5
17.3. The ContentHandlerFactory Interface.. 16
17.4. A Content Handler for the FITS Image Format.. 19

Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

Chapter 17. Content Handlers

Content handlers are one of the ideas that got developers excited about Java in the first place. At the time that Java was first released, Netscape, NCSA, Spyglass, and a few other combatants were fighting a battle over who would control the standards for web browsing. One of the battlegrounds was different browsers' ability to handle various kinds of files. The first browsers understood only HTML. The next generation understood HTML and GIF. JPEG support was soon added. The intensity of this battle meant that new versions of browsers were released every couple of weeks. Netscape made the first attempt to break this infinite loop by introducing plug-ins in Navigator 2.0. Plug-ins are platform-dependent browser extenders written in C that add the ability to view new content types such as Adobe PDF and VRML. However, plug-ins have drawbacks. Each new content type requires the user to download and install a new plug-in, if indeed the right plug-in is even available for the user's platform. To keep up, users had to expend bandwidth and time downloading new browsers and plug-ins, each of which fixed a few bugs and added a few new features.

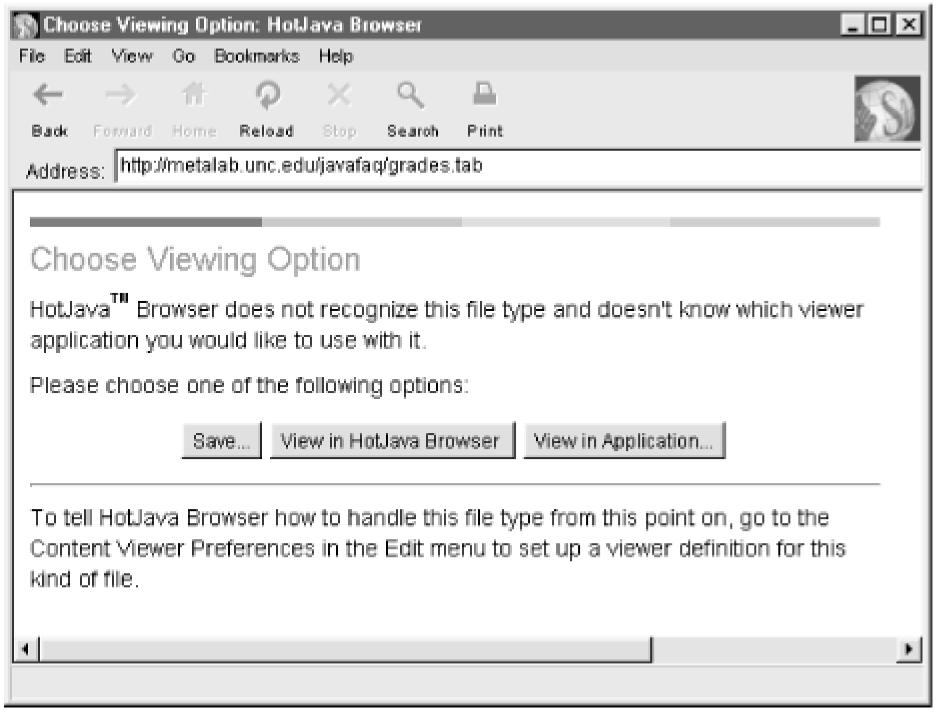
The Java team saw a way around this dilemma. Their idea was to use Java to download only the parts of the program that had to be updated rather than the entire browser. Furthermore, when the user encountered a web page that used a new content type, the browser could automatically download the code that was needed to view that content type. The user wouldn't have to stop, FTP a plug-in, quit the browser, install the plug-in, restart the browser, and reload the page. The mechanism that the Java team envisioned was the *content handler*. Each new data type that a web site wanted to serve would be associated with a content handler written in Java. The content handler would be responsible for parsing the content and displaying it to the user in the web browser window. The abstract class that content handlers for specific data types such as PNG or RTF would extend was `java.net.ContentHandler`. James Gosling and Henry McGilton described this scenario in 1996:

> HotJava's dynamic behavior is also used for understanding different types of objects. For example, most Web browsers can understand a small set of image formats (typically GIF, X11 pixmap, and X11 bitmap). If they see some other type, they have no way to deal with it. HotJava, on the other hand, can dynamically link the code from the host that has

the image, allowing it to display the new format. So, if someone invents a new compression algorithm, the inventor just has to make sure that a copy of its Java code is installed on the server that contains the images they want to publish; they don't have to upgrade all the browsers in the world. HotJava essentially upgrades itself on the fly when it sees this new type. (James Gosling and Henry McGilton, *The Java Language Environment, A White Paper*, May 1996, http://java.sun.com/docs/white/langenv/HotJava.doc1.html)

Unfortunately, content handlers never really made it out of Sun's white papers into shipping software. The `ContentHandler` class still exists in the standard library, and it has some uses in custom applications. However, neither HotJava nor any other web browser actually uses it to display content. When HotJava downloads an HTML page or a bitmapped image, it handles it with hardcoded routines that process that particular kind of data. When HotJava encounters an unknown content type, it simply asks the user to locate a helper application that can display the file, almost exactly like a traditional web browser such as Netscape Navigator or Internet Explorer, as xref linkend="ch17-35419"/ proves. The promise of dynamically extensible web browsers automatically downloading content handlers for new data types as they encounter them was never realized. Perhaps the biggest problem was that the `ContentHandler` class was too generic, providing too little information about what kind of object was being downloaded and how it should be displayed.

Figure 17-1. HotJava's reaction to an unexpected content type, even though a content handler for this type is installed


Choose Viewing Option: HotJava Browser
File Edit View Go Bookmarks Help
Back Forward Home Reload Stop Search Print
Address: http://metalab.unc.edu/javafaq/grades.tab
Save...
View in HotJava Browser
View in Application...

Choose Viewing Option

HotJava™ Browser does not recognize this file type and doesn't know which viewer application you would like to use with it.

Please choose one of the following options:

To tell HotJava Browser how to handle this file type from this point on, go to the Content Viewer Preferences in the Edit menu to set up a viewer definition for this kind of file.

A much more robust and better thought-out content handler mechanism is now available under the name JavaBeans Activation Framework. This is a standard extension to Java that provides the necessary API for deciding what to do with arbitrary datatypes at runtime. However, JAF has not yet been used inside web browsers or even widely adopted, although that shouldn't stop you from using it inside your own applications if you find it useful. See http://java.sun.com/beans/glasgow/jaf.html for more details.

17.1. What Is a Content Handler?

A content handler is an instance of a subclass of `java.net.ContentHandler`:

```
public abstract class ContentHandler extends Object
```

The SAX2 API for XML parsing defines a completely separate interface named `ContentHandler`. This has nothing to do with the content handlers we're discussing in this chapter.

This class knows how to take a `URLConnection` and a MIME type and turn the data coming from the `URLConnection` into a Java object of an appropriate type. Thus, a content handler allows a program to understand new kinds of data. Since Java lowers the bar for writing code below what's needed to write a browser or a Netscape plug-in, the theory is that many different web sites can write custom handlers, rather than having to rely on the overworked browser manufacturers.

Java can already download classes from the Internet. Thus, there isn't much magic to getting it to download a class that can understand a new content type. A content handler is just a *.class* file like any other. The magic is all inside the browser, which knows when and where to request a *.class* file to view a new content type. Of course, some browsers are more magical than others. Currently, the only way to make this work in a browser is in conjunction with an applet that knows how to request the content handler explicitly. It can also be used—in fact, more easily—in a standalone application that ignores browsers completely.

Specifically, a content handler reads data from a `URLConnection` and constructs an object appropriate for the content type from the data. Each subclass of `ContentHandler` handles a specific MIME type and subtype, such as `text/plain` or `image/gif`. Thus, an `image/gif` content handler returns a `URLImageSource` object (a class that implements the `ImageProducer` interface), while a `text/plain` content handler returns a `String`. A database content handler might return a `java.sql.ResultSet` object. An `application/x-macbinhex40` content handler might return a `BinhexDecoder` object written by the same programmer who wrote the `application/x-macbinhex40` content handler.

Content handlers are intimately tied to protocol handlers. In the previous chapter, the `getContent()` method of the `URLConnection` class returned an `InputStream` that fed the data from the server to the client. This works for simple protocols that only return ASCII text, such as finger, whois, and daytime. However, returning an input stream doesn't work well for protocols such as FTP, gopher, and HTTP, which can return a lot of different content types, many of which can't be understood as a stream of ASCII text. For protocols like these, `getContent( )` needs to check the MIME type and use the `createContentHandler()` method of the application's `ContentHandlerFactory` to produce a matching content handler. Once a `ContentHandler` exists, the `URLConnection`'s `getContent( )` method calls the `ContentHandler`'s `getContent( )` method, which creates the Java object to be returned. Outside of the `getContent()` method of a `URLConnection`, you rarely, if ever, call any `ContentHandler` method. Applications should never call the methods of a `ContentHandler` directly. Instead, they should use the `getContent( )` method of `URL` or `URLConnection`.

An object that implements the `ContentHandlerFactory` interface is responsible for choosing the right `ContentHandler` to go with a MIME type. The static `URLConnection.setContentHandlerFactory( )` method installs a `ContentHandlerFactory` in a program. Only one `ContentHandlerFactory` may be chosen during the lifetime of an application. When a program starts running, there is no `ContentHandlerFactory`; that is, the `ContentHandlerFactory` is `null`.

When there is no factory, Java looks for content handler classes with the name *`type.subtype`*, where *`type`* is the MIME type of the content (e.g., `text`) and *`subtype`* is the MIME subtype (e.g., `html`). It looks for these classes first in any packages named by the `java.content.handler.pkgs` property, then in the `sun.net.www.content` package. The `java.content.handler.pkgs` property should contain a list of package prefixes separated from each other by a vertical bar (|). This is similar to how Java finds protocol handlers. For example, if the `java.content.handler.pkgs` property has the value `com.macfaq.net.www.content|org.cafeaulait.content` and a program needs a content handler for application/xml files, it first tries to instantiate `com.macfaq.net.www.content.application.xml`. If that fails, it next tries to instantiate `org.cafeaulait.content.application.xml`. If that fails, as a last resort, it tries to instantiate `sun.net.www.content.application.xml`. These conventions are also used to search for a content handler if a `ContentHandlerFactory` is installed but the `createContentHandler( )` method returns `null`.

To summarize, here's the sequence of events:

1. A `URL` object is created that points at some Internet resource.
2. The `URL`'s `getContent()` method is invoked.
3. The `getContent( )` method of the `URL` calls the `getContent( )` method of its underlying `URLConnection`.
4. The `URLConnection.getContent( )` method calls the nonpublic method `getContentHandler( )` to find a content handler for the MIME type and subtype.
5. `getContentHandler( )` checks to see whether it already has a handler for this type in its cache. If it does, that handler is returned to `getContent( )`. Thus, browsers won't download content handlers for common types such as `text/html` every time the user goes to a new web page.
6. If there wasn't an appropriate `ContentHandler` in the cache and the `ContentHandlerFactory` isn't `null`, `getContentHandler( )` calls the `ContentHandlerFactory`'s `createContentHandler( )` method to instantiate a new `ContentHandler`. If this is successful, the `ContentHandler` object is returned to `getContent( )`.
7. If the `ContentHandlerFactory` is `null` or `createContentHandler( )` fails to instantiate a new `ContentHandler`, Java looks for a content handler class named *`type.subtype`*, where *`type`* is the MIME type of the content and *`subtype`* is the MIME subtype in one of the packages named in the `java.content.handler.pkgs` system property. If a content handler is found, it is returned. Otherwise...
8. Java looks for a content handler class named `sun.net.www.content.`*`type.subtype`*. If it's found, it's returned. Otherwise, `createContentHandler( )` returns `null`.
9. If the `ContentHandler` object is not `null`, the `ContentHandler`'s `getContent( )` method is called. This method returns an object appropriate for the content type. If the `ContentHandler` is `null`, an `IOException` is thrown.
10. Either the returned object or the exception is passed up the call chain, eventually reaching the method that invoked `getContent ( )`.

You can affect this chain of events in three ways: first, by constructing a `URL` and calling its `getContent( )` method; second, by creating a new `ContentHandler` subclass that `getContent()` can use; and third, by installing a `ContentHandlerFactory` with `URLConnection.setContentHandlerFactory( )`, changing the way the application looks for content handlers.

17.2. The ContentHandler Class

A subclass of `ContentHandler` overrides the `getContent()` method to return an object that's the Java equivalent of the content. This method can be quite simple or quite complex, depending almost entirely on the complexity of the content type you're trying to parse. A `text/plain` content handler is quite simple; a `text/rtf` content handler would be very complex.

The `ContentHandler` class has only a simple noargs constructor:

```
public ContentHandler( )
```

Since `ContentHandler` is an abstract class, you never call its constructor directly, only from inside the constructors of subclasses.

The primary method of the class, albeit an abstract one, is `getContent( )`:

```
public abstract Object getContent(URLConnection uc) throws IOException
```

This method is normally called only from inside the `getContent( )` method of a `URLConnection` object. It is overridden in a subclass that is specific to the type of content being handled. `getContent( )` should use the `URLConnection`'s `InputStream` to create an object. There are no rules about what type of object a content handler should return. In general, this depends on what the application requesting the content expects. Content handlers for text-like content bundled with the JDK return some subclass of `InputStream`. Content handlers for images return `ImageProducer` objects.

The `getContent( )` method of a content handler does not get the full `InputStream` that the `URLConnection` has access to. The `InputStream` that a content handler sees should include only the content's raw data. Any MIME headers or other protocol-specific information that come from the server should be stripped by the `URLConnection` before it passes the stream to the `ContentHandler`. A `ContentHandler` is only responsible for content, not for any protocol overhead that may be present. The `URLConnection` should have already performed any necessary handshaking with the server and interpreted any headers it sends.

17.2.1. A Content Handler for Tab-Separated Values

To see how content handlers work, let's create a `ContentHandler` that handles the `text/tab-separated-values` content type. We aren't concerned with how the tab-separated values get to us. That's for a protocol handler to deal with. All a `ContentHandler` needs to know is the MIME type and format of the data.

Tab-separated values are produced by many database and spreadsheet programs. A tab-separated file may look something like this (tabs are indicated by arrows).

```
JPE Associates → 341 Lafayette Street, Suite 1025 →
New York → NY → 10012
O'Reilly & Associates → 103 Morris Street, Suite A →
Sebastopol → CA → 95472
```

In database parlance, each line is a *record*, and the data before each tab is a *field*. It is usually (though not necessarily) true that each field has the same meaning in each record. In the previous example, the first field is the company name.

The first question to ask is: what kind of Java object should we convert the tab- separated values to? The simplest and most general way to store each record is as an array of `String`s. Successive records can be collected in a `Vector`. In many applications, however, you have a great deal more knowledge about the exact format and meaning of the data than we do here. The more you know about the data you're dealing with, the better a `ContentHandler` you can write. For example, if you know that the data you're downloading represents U.S. addresses, you could define a class like this:

```
public class Address {

  private String name;
  private String street;
  private String city;
  private String state;
  private String zip;

}
```

This class would also have appropriate constructors and other methods to represent each record. In this example, we don't know anything about the data in advance, or how many records we'll have to store. Therefore, we will take the most general approach and convert each record into an array of strings, using a `Vector` to store each array until there are no more records. The `getContent ( )` method can return the `Vector` of `String` arrays.

Example 17-1 shows the code for such a `ContentHandler`. The full package-qualified name is `com.macfaq.net.www.content.text.tab_separated_values`. This unusual class name follows the naming convention for a content handler for the MIME type `text/tab-separated-values`. Since MIME types often contain hyphens, as in this example, a convention exists to replace these with the underscore (_). Thus `text/tab-separated-values` becomes `text.tab_separated_values`. To install this content handler, all that's needed is to put the compiled *.class* file somewhere the class loader can find it and set the `java.content.handler.pkgs` property to `com.macfaq.net.www.content`.

Example 17-1. A ContentHandler for text/tab-separated-values

```
package com.macfaq.net.www.content.text;

import java.net.*;
import java.io.*;
import java.util.*;
import com.macfaq.io.SafeBufferedReader  // From Chapter 4

public class tab_separated_values extends ContentHandler {

  public Object getContent(URLConnection uc) throws IOException {
    String theLine;
    Vector lines = new Vector( );

    InputStreamReader isr = new InputStreamReader(uc.getInputStream( ));
    SafeBufferedReader in = new SafeBufferedReader(isr);
    while ((theLine = in.readLine( )) != null) {
      String[] linearray = lineToArray(theLine);
      lines.addElement(linearray);
    }

    return lines;

  }

  private String[] lineToArray(String line)  {

    int numFields = 1;
    for (int i = 0; i < line.length( ); i++) {
      if (line.charAt(i) == '\t') numFields++;
    }
    String[] fields = new String[numFields];
    int position = 0;
    for (int i = 0; i < numFields; i++) {
      StringBuffer buffer = new StringBuffer( );
      while (position < line.length( ) && line.charAt(position) != '\t') {
        buffer.append(line.charAt(position));
        position++;
      }
      fields[i] = buffer.toString( );
      position++;
    }

    return fields;

  }
}
```

Example 17-1 has two methods. The private utility method `lineToArray( )` converts a tab-separated string into an array of strings. This method is for the private use of this subclass and is not required by the `ContentHandler` interface. The more complicated the content you're trying to parse, the more such methods your class will need. The `lineToArray( )` method begins by counting the number of tabs in the string. This sets the `numFields` variable to one more than the number of tabs. An array is created for the fields with the length `numFields`; a `for` loop fills the array with the strings between the tabs; and this array is returned.

You may have expected a `StringTokenizer` to split the line into parts. However, that class has unusual ideas about what makes up a token. In particular, it interprets multiple tabs in a row as a single delimiter. That is, it never returns an empty string as a token.

The `getContent( )` method starts by instantiating a `Vector`. Then it gets the `InputStream` from the `URLConnection` `uc` and chains this to an `InputStreamReader`, which is in turn chained to the `SafeBufferedReader` (introduced in Chapter 4) so `getContent( )` can read the array one line at a time in a `while` loop. Each line is fed to the `lineToArray( )` method, which splits it into a `String` array. This array is then added to the `Vector`. When no more lines are left, the loop exits and the `Vector` is returned.

17.2.2. Using Content Handlers

Now that you've written your first `ContentHandler`, let's see how to use it in a program. Files of MIME type `text/tab-separated-values` can be served by gopher servers, HTTP servers, FTP servers, and more. Let's assume you're retrieving a tab-separated-values file from an HTTP server. The filename should end with the *.tsv* or *.tab* extension so that the server knows it's a `text/tab-separated-values` file.

Not all servers are configured to support this type out of the box. Consult your server documentation to see how to set up a MIME-type mapping for your server. For instance, to configure my Apache server, I added these lines to my *.htaccess* file:

```
AddType text/tab-separated-values tab
AddType text/tab-separated-values tsv
```

You can test the web server configuration by connecting to port 80 of the web server with Telnet and requesting the file manually:

```
% telnet www.ibiblio.org 80
Trying 127.0.0.1...
Connected to www.ibiblio.org.
Escape character is '^]'.
GET /javafaq/addresses.tab HTTP 1.0

HTTP 1.0 200 OK
Date: Mon, 15 Nov 1999 18:36:51 GMT
Server: Apache/1.3.4 (Unix) PHP/3.0.6 mod_perl/1.17
Last-Modified: Thu, 04 Nov 1999 18:22:51 GMT
Content-type: text/tab-separated-values
Content-length: 163

JPE Associates 341 Lafayette Street, Suite 1025 New York NY 10012
O'Reilly & Associates 103 Morris Street, Suite A Sebastopol CA 95472
Connection closed by foreign host.
```

You're looking for a line that says `Content-type: text/tab-separated-values`. If you see a `Content-type` of `text/plain`, `application/octet-stream`, or some other value, or you don't see any `Content-type` at all, the server is misconfigured and must be fixed before you continue.

The application that uses the tab-separated-values content handler does not need to know about it explicitly. It simply has to call the `getContent( )` method of `URL` or `URLConnection` on a URL with a matching MIME type. Furthermore, the package where the content handler can be found has to be listed in the `java.content.handlers.pkg` property.

Example 17-2 is a class that downloads and prints a `text/tab-separated-values` file using the `ContentHandler` of Example 17-1. However, note that it does not import `com.macfaq.net.www.content.text` and never references the `tab_separated_values` class. It does explicitly add `com.macfaq.net.www.content` to the `java.content.handlers.pkgs` property because that's the simplest way to make sure this standalone program works. However, the lines that do this could be deleted if the property were set in a property file or from the command line.

Example 17-2. The tab-separated-values ContentTester class

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

public class TSVContentTester {

  private static void test(URL u) throws IOException {

    Object content = u.getContent( );
    Vector v = (Vector) content;
    for (Enumeration e = v.elements( ) ; e.hasMoreElements( ) ;) {
      String[] sa = (String[]) e.nextElement( );
      for (int i = 0; i < sa.length; i++) {
        System.out.print(sa[i] + "\t");
      }
      System.out.println( );
    }

  }

  public static void main (String[] args) {

    // If you uncomment these lines, then you don't have to
    // set the java.content.handler.pkgs property from the
    // command line or your properties files.

/*    String pkgs = System.getProperty("java.content.handler.pkgs", "");
    if (!pkgs.equals("")) {
      pkgs = pkgs + "|";
    }
    pkgs += "com.macfaq.net.www.content";
    System.setProperty("java.content.handler.pkgs", pkgs);  */

    for (int i = 0; i < args.length; i++) {
      try {
        URL u = new URL(args[i]);
        test(u);
      }
      catch (MalformedURLException ex) {
        System.err.println(args[i] + " is not a good URL");
      }
      catch (Exception ex) {
        ex.printStackTrace( );
      }
    }
  }
}
```

Here's how you run this program. The arrows indicate tabs:

```
% java -Djava.content.handler.pkgs=com.macfaq.net.www.content\
             TSVContentTester http://www.ibiblio.org/javafaq/addresses.tab
JPE Associates → 341 Lafayette Street, Suite 1025 → New York →
NY → 10012
O'Reilly & Associates → 103 Morris Street, Suite A →
Sebastopol → CA → 95472
```

17.2.3. Choosing Return Types

There is one overloaded variant of the `getContent( )` method in the `ContentHandler` class:

```
public Object getContent(URLConnection uc, Class[] classes) // Java 1.3
 throws IOException
```

The difference is the array of `java.lang.Class` objects passed as the second argument. This allows the caller to request that the content be returned as one of the types in the array and enables content handlers to support multiple types. For example, the `text/tab-separated-values` content handler could return data as a `Vector`, an array, a string, or an `InputStream`. One would be the default used by the single argument `getContent( )` method, while the others would be options that a client could request. If the client doesn't request any of the classes this `ContentHandler` knows how to provide, it returns `null`.

To call this method, the client invokes the method with the same arguments in a `URL` or `URLConnection` object. It passes an array of `Class` objects in the order it wishes to receive the data. Thus, if it prefers to receive a `String` but is willing to accept an `InputStream` and will take a `Vector` as a last resort, it puts `String.class` in the zeroth component of the array, `InputStream.class` in the first component of the array, and `Vector.class` in the last component of the array. Then it uses `instanceof` to test what was actually returned and either process it or convert it into the preferred type. For example:

```
Class[] requestedTypes = {String.class, InputStream.class,
 Vector.class};
Object content = url.getContent(requestedTypes);
if (content instanceof String) {
  String s = (String) content;
  System.out.println(s);
}
else if (content instanceof InputStream) {
  InputStream in = (InputStream) content;
  int c;
  while ((c = in.read( )) != -1) System.out.write(c);
}
else if (content instanceof Vector) {
  Vector v = (Vector) content;
  for (Enumeration e = v.elements( ) ; e.hasMoreElements( ) ;) {
    String[] sa = (String[]) e.nextElement( );
    for (int i = 0; i < sa.length; i++) {
      System.out.print(sa[i] + "\t");
    }
    System.out.println( );
  }
}
else {
  System.out.println("Unrecognized content type " + content.getClass( ));
}
```

To demonstrate this, let's write a content handler that can be used in association with the time protocol. Recall that the time protocol returns the current time at the server as a 4-byte, big-endian, unsigned integer giving the number of seconds since midnight, January 1, 1900, Greenwich Mean Time. There are several obvious candidates for storing this data in a Java content handler, including `java.lang.Long` (`java.lang.Integer` won't work since the unsigned value may overflow the bounds of an `int`), `java.util.Date`, `java.util.Calendar`, `java.lang.String`, and `java.io.InputStream`, which often works as a last resort. Example 17-3 provides all five options. There's no standard MIME type for the time format. We'll use `application` for the type to indicate that this is binary data and `x-time` for the subtype to indicate that this is a nonstandard extension type. It will be up to the time protocol handler to return the right content type.

Example 17-3. A time content handler

```
package com.macfaq.net.www.content.application;

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

public class x_time extends ContentHandler {

  public Object getContent(URLConnection uc) throws IOException {

    Class[] classes = new Class[1];
    classes[0] = Date.class;
    return this.getContent(uc, classes);

  }

  public Object getContent(URLConnection uc, Class[] classes)
```

```
     throws IOException {

      InputStream in = uc.getInputStream( );
      for (int i = 0; i < classes.length; i++) {
        if (classes[i] == InputStream.class) {
          return in;
        }
        else if (classes[i] == Long.class) {
          long secondsSince1900 = readSecondsSince1900(in);
          return new Long(secondsSince1900);
        }
        else if (classes[i] == Date.class) {
          long secondsSince1900 = readSecondsSince1900(in);
          Date time = shiftEpochs(secondsSince1900);
          return time;
        }
        else if (classes[i] == Calendar.class) {
          long secondsSince1900 = readSecondsSince1900(in);
          Date time = shiftEpochs(secondsSince1900);
          Calendar c = Calendar.getInstance( );
          c.setTime(time);
          return c;
        }
        else if (classes[i] == String.class) {
          long secondsSince1900 = readSecondsSince1900(in);
          Date time = shiftEpochs(secondsSince1900);
          return time.toString( );
        }
      }

      return null; // no requested type available

    }

    private long readSecondsSince1900(InputStream in)
     throws IOException {

      long secondsSince1900 = 0;
      for (int j = 0; j < 4; j++) {
        secondsSince1900 = (secondsSince1900 << 8) | in.read( );
      }
      return secondsSince1900;

    }

    private Date shiftEpochs(long secondsSince1900) {

      // The time protocol sets the epoch at 1900, the Java Date class
      //  at 1970. This number converts between them.
      long differenceBetweenEpochs = 2208988800L;

      long secondsSince1970 = secondsSince1900 - differenceBetweenEpochs;
      long msSince1970 = secondsSince1970 * 1000;
      Date time = new Date(msSince1970);
      return time;

    }
  }
```

Most of the work is performed by the second `getContent()` method, which checks to see whether it recognizes any of the classes in the `classes` array. If so, it attempts to convert the content into an object of that type. The `for` loop is arranged so that classes earlier in the array take precedence; that is, it first tries to match the first class in the array; next it tries to match the second class in the array; then the third class in the array; and so on. As soon as one class is matched, the method returns so later classes won't be matched even if they're an allowed choice.

Once a type is matched, a simple algorithm converts the four bytes that the time server sends into the right kind of object, either an `InputStream`, a `Long`, a `Date`, a `Calendar`, or a `String`. The `InputStream` conversion is trivial. The `Long` conversion is one of those times when it seems a little inconvenient that primitive data types aren't objects. Although you can convert to and return any object type, you can't convert to and return a primitive data type like `long`, so we return the type wrapper class `Long` instead. The `Date` and `Calendar` conversions require shifting the origin of the time from January 1, 1900 to January 1, 1970 and changing the units from seconds to milliseconds, as discussed in Chapter 9. Finally, the conversion to a `String` simply converts to a `Date` and then invokes the `Date` object's `toString( )` method.

While it would be possible to configure a web server to send data of MIME type `application/x-time`, this class is really designed to be used by a custom protocol handler. This handler would know not only how to speak the time protocol, but also how to return `application/x-time` from the `getContentType( )` method. Example 17-4 and Example 17-5 demonstrate such a protocol handler. It assumes that time URLs look like *time://vision.poly.edu:3737/*.

Example 17-4. The URLConnection for the time protocol handler

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

import java.net.*;
import java.io.*;
import com.macfaq.net.www.content.application.*;

public class TimeURLConnection extends URLConnection {

  private Socket connection = null;
  public final static int DEFAULT_PORT = 37;

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

  public String getContentType( ) {
    return "application/x-time";
  }

  public Object getContent( ) throws IOException {
    ContentHandler ch = new x_time( );
    return ch.getContent(this);
  }

  public Object getContent(Class[] classes) throws IOException {
    ContentHandler ch = new x_time( );
    return ch.getContent(this, classes);
  }

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

```
    public synchronized void connect( ) throws IOException {

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

In general, it should be enough for the protocol handler to simply know or be able to deduce the correct MIME content type. However, in a case like this, where both content and protocol handlers must be provided, you can tie them a little more closely together by overriding `getContent ( )` as well. This allows you to avoid messing with the `java.content.handler.pkgs` property or installing a `ContentHandlerFactory`. You will still need to set the `java.protocolhandler.pkgs` property to point to your package or install a `URLStreamHandlerFactory`, however. Example 17-5 is a simple `URLStreamHandler` for the time protocol handler.

Example 17-5. The URLStreamHandler for the time protocol handler

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

import java.net.*;
import java.io.*;
public class Handler extends URLStreamHandler {

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

We could install the time protocol handler into HotJava as we did with protocol handlers in the previous chapter. However, even if we place the time content handler in HotJava's class path, HotJava won't use it. Consequently, I've written a simple standalone application, shown in Example 17-6, that uses these protocol and content handlers to tell the time. Notice that it does not need to import or directly refer to any of the classes involved. It simply lets the `URL` find the right content handler.

Example 17-6. URLTimeClient

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

public class URLTimeClient {

  public static void main(String[] args) {

    System.setProperty("java.protocol.handler.pkgs",
     "com.macfaq.net.www.protocol");

    try {
      // You can replace this with your own time server
      URL u = new URL("time://tock.usno.navy.mil/");
      Class[] types = {String.class, Date.class,
       Calendar.class, Long.class};
      Object o = u.getContent(types);
      System.out.println(o);
    }
    catch (IOException ex) {
     // Let's see what went wrong
     ex.printStackTrace( );
    }
  }
}
```

Here's a sample run:

```
D:\JAVA\JNP3\examples\17>java URLTimeClient
Mon Aug 23 21:30:34 EDT 2004
```

In this case, a `String` object was returned. This was the first choice of `URLTimeClient` but the last choice of the content handler. The client choice always takes precedence.

17.3. The ContentHandlerFactory Interface

A `ContentHandlerFactory` defines the rules for where `ContentHandler` classes are stored. Create a class that implements `ContentHandlerFactory` and give this class a `createContentHandler( )` method that knows how to instantiate the right `ContentHandler`. The `createContentHandler( )` method should return `null` if it can't find a `ContentHandler` appropriate for a MIME type; `null` signals Java to look for `ContentHandler` classes in the default locations. When the application starts, call the `URLConnection`'s `setContentHandlerFactory( )` method to set the `ContentHandlerFactory`. This method may be called only once in the lifetime of an application.

17.3.1. The createContentHandler( ) Method

Just as the `createURLStreamHandler( )` method of the `URLStreamHandlerFactory` interface was responsible for finding and loading the appropriate protocol handler, so too the `createContentHandler()` method of the `ContentHandlerFactory` interface is responsible for finding and loading the appropriate `ContentHandler` given a MIME type:

```
public abstract ContentHandler createContentHandler(String mimeType)
```

This method should be called only by the `getContent()` method of a `URLConnection` object. For instance, Example 17-7 is a `ContentHandlerFactory` that knows how to find the right handler for the `text/tab-separated-values` content handler of Example 17-1.

Example 17-7. TabFactory

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

import java.net.*;

public class TabFactory implements ContentHandlerFactory {

  public ContentHandler createContentHandler(String mimeType)) {

    if (mimeType.equals("text/tab-separated-values") {
      return new com.macfaq.net.www.content.text.tab_separated_values( );
    }
    else {
      return null; // look for the handler in the default locations
    }
  }
}
```

This factory knows how to find only one kind of content handler, but there's no limit to how many a factory can know about. For example, this `createContentHandler( )` method also suggests handlers for `application/x-time`, `text/plain`, `video/mpeg`, and `model/vrml`. Notice that when you're using a `ContentHandlerFactory`, you don't necessarily have to stick to standard naming conventions for `ContentHandler` subclasses:

```
public ContentHandler createContentHandler(String mimeType)) {

  if (mimeType.equals("text/tab-separated-values") {
      return new com.macfaq.net.www.content.text.tab_separated_values( );
  }
  else if (mimeType.equals("application/x-time") {
    return new com.macfaq.net.www.content.application.x_time( );
  }
  else if (mimeType.equals("text/plain") {
    return new sun.net.www.content.text.plain( );
  }
  if (mimeType.equals("video/mpeg") {
    return new com.macfaq.video.MPEGHandler( );
  }
  if (mimeType.equals("model/vrml") {
    return new com.macfaq.threed.VRMLModel( );
  }
  else {
    return null; // look for the handler in the default locations
  }
}
```

17.3.2. Installing Content Handler Factories

A `ContentHandlerFactory` is installed in an application using the static `URLConnection.setContentHandlerFactory()` method:

```
public static void setContentHandlerFactory(ContentHandlerFactory fac)
```

Note that this method is in the `URLConnection` class, not the `ContentHandler` class. It may be invoked at most once during any run of an application. It throws an `Error` if it is called a second time.

Using a `ContentHandlerFactory` such as the `TabFactory` in Example 17-5, it's possible to write a standalone application that can automatically load the tab-separated-values content handler and that runs without any major hassles with the class path. Example 17-8 is such a program. However, as with most other `setFactory( )` methods, untrusted, remotely loaded code such as an applet will generally not be allowed to set the content handler factory. Attempting to do so will throw a `SecurityException`. Consequently, installing new content handlers in applets pretty much requires directly accessing the `getContent( )` method of the `ContentHandler` subclass itself. Ideally, this shouldn't be necessary, but until Sun provides better support for downloadable content handlers in browsers, we're stuck with it.

Example 17-8. TabLoader that uses a ContentHandlerFactory

```
import java.io.*;
import java.net.*;
import java.util.*;
import com.macfaq.net.www.content.*;

public class TabLoader {

  public static void main (String[] args) {

    URLConnection.setContentHandlerFactory(new TabFactory( ));

    for (int i = 0; i < args.length; i++) {
      try {
        URL u = new URL(args[i]);
        Object content = u.getContent( );
        Vector v = (Vector) content;
        for (Enumeration e = v.elements( ) ; e.hasMoreElements( ) ;) {
          String[] sa = (String[]) e.nextElement( );
          for (int j = 0; j < sa.length; j++) {
            System.out.print(sa[j] + "\t");
          }
          System.out.println( );
        }
      }
      catch (MalformedURLException ex) {
        System.err.println(args[i] + " is not a good URL");
      }
      catch (Exception ex) {
        ex.printStackTrace( );
      }
    }
  }
}
```

Here's a typical run. As usual, tabs are indicated by arrows:

```
% java TabLoader http://www.ibiblio.org/javafaq/addresses.tab
JPE Associates → 341 Lafayette Street, Suite 1025 → New York →
NY → 10012
O'Reilly & Associates → 103 Morris Street, Suite A →
Sebastopol → CA → 95472
```

17.4. A Content Handler for the FITS Image Format

That's really all there is to content handlers. As one final example, I'll show you how to write a content handler for image files. This kind of handler differs from the text-based content handlers you've already seen in that they generally produce an object that implements the `java.awt.ImageProducer` interface rather than an `InputStream` object. The specific example we'll choose is the Flexible Image Transport System (FITS) format in common use among astronomers. FITS files are grayscale, bitmapped images with headers that determine the bit depth of the picture, the width and the height of the picture, and the number of pictures in the file. Although FITS files often contain several images (typically pictures of the same thing taken at different times), in this example we look at only the first image in a file. For more details about the FITS format and how to handle FITS files, see *The Encyclopedia of Graphics File Formats* by James D. Murray and William vanRyper (O'Reilly).

There are a few key things you need to know to process FITS files. First, FITS files are broken up into blocks of exactly 2,880 bytes. If there isn't enough data to fill a block, it is padded with spaces at the end. Each FITS file has two parts, the header and the primary data unit. The header occupies an integral number of blocks, as does the primary data unit. If the FITS file contains extensions, there may be additional data after the primary data unit, but we ignore that here. Any extensions that are present will not change the image contained in the primary data unit.

The header begins in the first block of the FITS file. It may occupy one or more blocks; the last block may be padded with spaces at the end. The header is ASCII text. Each line of the header is exactly 80 bytes wide; the first eight characters of each header line contain a keyword, which is followed by an equals sign (character 9), followed by a space (10). The keyword is padded on the right with spaces to make it eight characters long. Columns 11 through 30 contain a value; the value may be right-justified and padded on the left with spaces if necessary. The value may be an integer, a floating point number, a `T` or an `F` signifying the boolean values true and false, or a string delimited with single quotes. A comment may appear in columns 31 through 80; comments are separated from the value of a field by a slash (/). Here's a simple header taken from a FITS image produced by K. S. Balasubramaniam using the Dunn Solar Telescope at the National Solar Observatory in Sunspot, New Mexico (http://www.sunspot.noao.edu/):

```
SIMPLE  =                    T /
BITPIX  =                   16 /
NAXIS   =                    2 /
NAXIS1  =                  242 /
NAXIS2  =                  252 /
DATE    = '19 Aug 1996'        /
TELESC  = 'NSO/SP - VTT'       /
IMAGE   = 'Continuum'          /
COORDS  = 'N29.1W34.2'         /
OBSTIME = '13:59:00 UT'        /
END
```

Every FITS file begins with the keyword SIMPLE. This keyword always has the value `T`. If this isn't the case, the file is not valid. The second line of a FITS file always has the keyword BITPIX, which tells you how the data is stored. There are five possible values for BITPIX, four of which correspond exactly to Java primitive data types. The most common value of BITPIX is 16, meaning that there are 16 bits per pixel, which is equivalent to a Java `short`. A BITPIX of 32 is a Java `int`. A BITPIX of -32 means that each pixel is represented by a 32-bit floating point number (equivalent to a Java `float`); a BITPIX of -64 is equivalent to a Java `double`. A BITPIX of 8 means that 8 bits are used to represent each pixel; this is similar to a Java `byte`, except that FITS uses unsigned bytes ranging from 0 to 255; Java's `byte` data type is signed, taking values that range from -128 to 127.

The remaining keywords in a FITS file may appear in any order. They are *not* necessarily in the order shown here. In our FITS content handler, we first read all the keywords into a `Hashtable` and then extract the ones we want by name.

The NAXIS header specifies the number of axes (that is, the dimension) of the primary data array. A NAXIS value of one identifies a one-dimensional image. A NAXIS value of two indicates a normal two-dimensional rectangular image. A NAXIS value of three is called a *data cube* and generally means the file contains a series of pictures of the same object taken at different moments in time. In other words, time is the third dimension. On rare occasions, the third dimension can represent depth: i.e., the file contains a true three-dimensional image. A NAXIS of four means the file contains a sequence of three-dimensional pictures taken at different moments in time. Higher values of NAXIS, while theoretically possible, are rarely seen in practice. Our example is going to look at only the first two-dimensional image in a file.

The NAXIS*n* headers (where *n* is an integer ranging from 1 to NAXIS) give the length of the image in pixels along that dimension. In this example, NAXIS1 is 242, so the image is 242 pixels wide. NAXIS2 is 252, so this image is 252 pixels high. Since FITS images are normally pictures of astronomical bodies like the sun, it doesn't really matter if you reverse width and height. All FITS images contain the SIMPLE, BITPIX, END, and NAXIS keywords, plus a series of NAXIS*n* keywords. These keywords all provide information that is essential for displaying the image.

The next five keywords are specific to this file and may not be present in other FITS files. They give meaning to the image, although they are not needed to display it. The DATE keyword says this image was taken on August 19, 1996. The TELESC keyword says this image was taken by the Vacuum Tower Telescope (VTT) at the National Solar Observatory (NSO) on Sacramento Peak (SP). The IMAGE keyword says that this is a picture of the white light continuum; images taken through spectrographs might look at only a particular wavelength in the spectrum. The COORDS keyword gives the latitude and longitude of the telescope. Finally, the OBSTIME keyword says this image was taken at 1:59 P.M. Universal Time (essentially, Greenwich Mean Time). There are many more optional headers that don't appear in this example. Like the five discussed here, the remaining keywords may help someone interpret an image, but they don't provide the information needed to display it.

The keyword END terminates the header. Following the END keyword, the header is padded with spaces so that it fills a 2,880-byte block. A header may take up more than one 2,880-byte block, but it must always be padded to an integral number of blocks.

The image data follows the header. How the image is stored depends on the value of BITPIX, as explained earlier. Fortunately, these data types are stored in formats (big-endian, two's complement) that can be read directly with a `DataInputStream`. The exact meaning of each number in the image data is completely file-dependent. More often than not, it's the number of electrons that were collected in a specific time interval by a particular pixel in a charge coupled device (CCD); in older FITS files, the numbers could represent the value read from photographic film by a densitometer. However, the unifying theme is that larger numbers represent brighter light. To interpret these numbers as a grayscale image, we map the smallest value in the data to pure black, the largest value in the data to pure white, and scale all intermediate values appropriately. A general-purpose FITS reader cannot interpret the numbers as anything except abstract brightness levels. Without scaling, differences tend to get washed out. For example, a dark spot on the Sun tends to be about 4,000K. That is dark compared to the normal solar surface temperature of 6,000K, but considerably brighter than anything you're likely to see on the surface of the Earth.

Example 17-9 is a FITS content handler. FITS files should be served with the MIME type `image/x-fits`. This is almost certainly not included in your server's default MIME-type mappings, so make sure to add a mapping between files that end in *. fit*, *.fts*, or *.fits* and the MIME type `image/x-fits`.

Example 17-9. An x-fits content handler

```
package com.macfaq.net.www.content.image;

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

public class x_fits extends ContentHandler {

  public Object getContent(URLConnection uc) throws IOException {

    int width = -1;
    int height = -1;
    int bitpix = 16;
    int[] data = null;
    int naxis = 2;
    Hashtable header = null;

    DataInputStream dis = new DataInputStream(uc.getInputStream( ));
    header = readHeader(dis);

    bitpix = getIntFromHeader("BITPIX  ", -1, header);
    if (bitpix  <= 0) return null;
    naxis = getIntFromHeader("NAXIS   ", -1, header);
    if (naxis  < 1) return null;
    width = getIntFromHeader("NAXIS1  ", -1, header);
    if (width  <= 0) return null;
    if (naxis == 1) height = 1;
    else height = getIntFromHeader("NAXIS2  ", -1, header);
    if (height  <= 0) return null;

    if (bitpix == 16) {
      short[] theInput =  new short[height * width];
      for (int i = 0; i < theInput.length; i++) {
        theInput[i] = dis.readShort( );
      }
      data = scaleArray(theInput);
    }
    else if (bitpix == 32) {
      int[] theInput =  new int[height * width];
      for (int i = 0; i < theInput.length; i++) {
        theInput[i] = dis.readInt( );
      }
      data = scaleArray(theInput);
   }
   else if (bitpix == 64) {
     long[] theInput =  new long[height * width];
     for (int i = 0; i < theInput.length; i++) {
       theInput[i] = dis.readLong( );
     }
     data = scaleArray(theInput);
   }
   else if (bitpix == -32) {
      float[] theInput =  new float[height * width];
      for (int i = 0; i < theInput.length; i++) {
        theInput[i] = dis.readFloat( );
      }
      data = scaleArray(theInput);
    }
    else if (bitpix == -64) {
      double[] theInput =  new double[height * width];
      for (int i = 0; i < theInput.length; i++) {
        theInput[i] = dis.readDouble( );
      }
      data = scaleArray(theInput);
    }
    else {
      System.err.println("Invalid BITPIX");
      return null;
    } // end if-else-if

    return new  MemoryImageSource(width, height, data, 0, width);

  }  // end getContent

  private Hashtable readHeader(DataInputStream dis)
   throws IOException {

    int blocksize = 2880;
    int fieldsize = 80;
    String key, value;
    int linesRead = 0;

    byte[] buffer = new byte[fieldsize];

    Hashtable header = new Hashtable( );
    while (true) {
      dis.readFully(buffer);
      key = new String(buffer, 0, 8, "ASCII");
      linesRead++;
      if (key.substring(0, 3).equals("END")) break;
      if (buffer[8] != '=' || buffer[9] != ' ') continue;
      value = new String(buffer, 10, 20, "ASCII");
      header.put(key, value);
    }
    int linesLeftToRead
     = (blocksize - ((linesRead * fieldsize) % blocksize))/fieldsize;
    for (int i = 0; i < linesLeftToRead; i++) dis.readFully(buffer);

    return header;

  }

  private int getIntFromHeader(String name, int defaultValue,
   Hashtable header) {

    String s = "";
    int result = defaultValue;

    try {
      s = (String) header.get(name);
    }
    catch (NullPointerException ex) {
      return defaultValue;
    }
    try {
      result = Integer.parseInt(s.trim( ));
    }
    catch (NumberFormatException ex) {
      System.err.println(ex);
      System.err.println(s);
      return defaultValue;
    }

    return result;

  }

  private int[] scaleArray(short[] theInput) {

    int data[] = new int[theInput.length];
    int max = 0;
    int min = 0;
    for (int i = 0; i < theInput.length; i++) {
      if (theInput[i] > max) max = theInput[i];
      if (theInput[i] < min) min = theInput[i];
    }
    long r = max - min;
    double a = 255.0/r;
    double b = -a * min;
    int opaque = 255;
    for (int i = 0; i < data.length; i++) {
      int temp = (int) (theInput[i] * a + b);
      data[i] =  (opaque << 24)  | (temp << 16)  | (temp << 8) | temp;
    }
    return data;

  }

  private int[] scaleArray(int[] theInput) {

    int data[] = new int[theInput.length];
    int max = 0;
    int min = 0;
    for (int i = 0; i < theInput.length; i++) {
      if (theInput[i] > max) max = theInput[i];
      if (theInput[i] < min) min = theInput[i];
    }
    long r = max - min;
    double a = 255.0/r;
    double b = -a * min;
    int opaque = 255;
    for (int i = 0; i < data.length; i++) {
      int temp = (int) (theInput[i] * a + b);
      data[i] =  (opaque << 24)  | (temp << 16)  | (temp << 8) | temp;
    }
    return data;

  }

  private int[] scaleArray(long[] theInput) {

    int data[] = new int[theInput.length];
    long max = 0;
    long min = 0;
    for (int i = 0; i < theInput.length; i++) {
      if (theInput[i] > max) max = theInput[i];
      if (theInput[i] < min) min = theInput[i];
    }
    long r = max - min;
    double a = 255.0/r;
    double b = -a * min;
    int opaque = 255;
    for (int i = 0; i < data.length; i++) {
      int temp = (int) (theInput[i] * a + b);
      data[i] =  (opaque << 24)  | (temp << 16)  | (temp << 8) | temp;
    }
    return data;

  }
```

```
      private int[] scaleArray(double[] theInput) {

        int data[] = new int[theInput.length];
        double max = 0;
        double min = 0;
        for (int i = 0; i < theInput.length; i++) {
          if (theInput[i] > max) max = theInput[i];
          if (theInput[i] < min) min = theInput[i];
        }
        double r = max - min;
        double a = 255.0/r;
        double b = -a * min;
        int opaque = 255;
        for (int i = 0; i < data.length; i++) {
          int temp = (int) (theInput[i] * a + b);
          data[i] =  (opaque << 24)  | (temp << 16)  | (temp << 8) | temp;
        }
        return data;

      }

      private int[] scaleArray(float[] theInput) {

        int data[] = new int[theInput.length];
        float max = 0;
        float min = 0;
        for (int i = 0; i < theInput.length; i++) {
          if (theInput[i] > max) max = theInput[i];
          if (theInput[i] < min) min = theInput[i];
        }
        double r = max - min;
        double a = 255.0/r;
        double b = -a * min;
        int opaque = 255;
        for (int i = 0; i < data.length; i++) {
          int temp = (int) (theInput[i] * a + b);
          data[i] =  (opaque << 24)  | (temp << 16)  | (temp << 8) | temp;
        }
        return data;
      }
    }
```

The key method of the `x_fits` class is `getContent( )`; it is the one method that the `ContentHandler` class requires subclasses to implement. The other methods in this class are all utility methods that help to break up the program into easier-to-digest chunks. `getContent ( )` is called by a `URLConnection`, which passes a reference to itself in the argument `uc`. The `getContent()` method reads data from that `URLConnection` and uses it to construct an object that implements the `ImageProducer` interface. To simplify the task of creating an `ImageProducer`, we create an array of image data and use a `MemoryImageSource` object, which implements the `ImageProducer` interface, to convert that array into an image. `getContent( )` returns this `MemoryImageSource`.

`MemoryImageSource` has several constructors. The one invoked here requires us to provide the width and height of the image, an array of integer values containing the RGB data for each pixel, the offset of the start of that data in the array, and the number of pixels per line in the array:

```
   public MemoryImageSource(int width, int height, int[] pixels,
    int offset, int scanlines);
```

The width, height, and pixel data can be read from the header of the FITS image. Since we are creating a new array to hold the pixel data, the offset is zero and the scanlines are the width of the image.

Our content handler has a utility method called `readHeader()` that reads the image header from `uc`'s `InputStream`. This method returns a `Hashtable` containing the keywords and their values as `String` objects. Comments are thrown away. `readHeader( )` reads 80 bytes at a time, since that's the length of each field. The first eight bytes are transformed into the `String` key. If there is no key, the line is a comment and is ignored. If there is a key, then the eleventh through thirtieth bytes are stored in a `String` called `value`. The key-value pair is stored in the `Hashtable`. This continues until the END keyword is spotted. At this point, we break out of the loop and read as many lines as necessary to finish the block. (Recall that the header is padded with spaces to make an integral multiple of 2,880.) Finally, `readHeader()` returns the `Hashtable header`.

After the header has been read into the `Hashtable`, the `InputStream` is now pointing at the first byte of data. However, before we're ready to read the data, we must extract the height, width, and bits per pixel of the primary data unit from the header. These are all integer values, so to simplify the code we use the `getIntFromHeader(String name, int defaultValue, Hashtable header)` method. This method takes as arguments the name of the header whose value we want (e.g., BITPIX), a default value for that header, and the `Hashtable` that contains the header. This method retrieves the value associated with the string `name` from the `Hashtable` and casts the result to a `String` object—we know this cast is safe because we put only `String` data into the `Hashtable`. This `String` is then converted to an `int` using `Integer.parseInt(s.trim( ))`; we then return the resulting `int`. If an exception is thrown, `getIntFromHeader( )` returns the `defaultValue` argument instead. In this content handler, we use an impossible flag value (-1) as the default to indicate that `getIntFromHeader( )` failed.

`getContent( )` uses `getIntFromHeader()` to retrieve four crucial values from the header: NAXIS, NAXIS1, NAXIS2, and BITPIX. NAXIS is the number of dimensions in the primary data array; if it is greater than or equal to two, we read the width and height from NAXIS1 and NAXIS2. If there are more than two dimensions, we still read a single two-dimensional frame from the data. A more advanced FITS content handler might read subsequent frames and include them below the original image or display the sequence of images as an animation. If NAXIS is one, the width is read from NAXIS1 and the height is set to one. (A FITS file with NAXIS as one would typically be produced from observations that used a one-dimensional CCD.) If NAXIS is less than one, there's no image data at all, so we return `null`.

Now we are ready to read the image data. The data can be stored in one of five formats, depending on the value of BITPIX: unsigned bytes, `shorts`, `ints`, `floats`, or `doubles`. This is where the lack of generics that can handle primitive types makes coding painful: we need to repeat the algorithm for reading data five times, once for each of the five possible data types. In each case, the data is first read from the stream into an array of the appropriate type called `theInput`. Then this array is passed to the `scaleArray( )` method, which returns a scaled array. `scaleArray( )` is an overloaded method that reads the data in `theInput` and copies the data into the `int` array `theData`, while scaling the data to fall from 0 to 255; there is a different version of `scaleArray( )` for each of the five data types it might need to handle. Thus, no matter what format the data starts in, it becomes an `int` array with values from 0 to 255. This data now needs to be converted into grayscale RGB values. The standard 32-bit RGB color model allows 256 different shades of gray, ranging from pure black to pure white; 8 bits are used to represent opacity, usually called "alpha". To get a particular shade of gray, the red, green, and blue bytes of an RGB triple should all be set to the same value, and the alpha value should be 255 (fully opaque). Thinking of these as four byte values, we need colors like 255.127.127.127 (medium gray) or 255.255.255.255 (pure white). These colors are produced by the lines:

```
int temp = (int) (theInput[i] * a + b);
theData[i] =  (opaque << 24)  | (temp << 16)  | (temp << 8) | temp;
```

Once it has converted every pixel in `theInput[]` into a 32-bit color value and stored the result in `theData[]`, `scaleArray( )` returns `theData`. The only thing left for `getContent( )` to do is feed this array, along with the header values previously retrieved, into the `MemoryImageSource` constructor and return the result.

This FITS content handler has one glaring problem. The image has to be completely loaded before the method returns. Since FITS images are quite literally astronomical in size, loading the image can take a significant amount of time. It would be better to create a new class for FITS images that implements the `ImageProducer` interface and into which the data can be streamed asynchronously. The `ImageConsumer` that eventually displays the image can use the methods of `ImageProducer` to determine when the height and width are available, when a new scanline has been read, when the image is completely loaded or errored out, and so on. `getContent( )` would spawn a separate thread to feed the data into the `ImageProducer` and would return almost immediately. However, a FITS `ImageProducer` would not be able to take significant advantage of progressive loading because the file format doesn't unambiguously define what each data value means; before we can generate RGB pixels, we must read all of the data and find the minimum and maximum values.

Example 17-10 is a simple `ContentHandlerFactory` that recognizes FITS images. For all types other than `image/x-fits`, it returns `null` so that the default locations will be searched for content handlers.

Example 17-10. The FITS ContentHandlerFactory

```
import java.net.*;

public class FitsFactory implements ContentHandlerFactory {

  public ContentHandler createContentHandler(String mimeType) {

    if (mimeType.equalsIgnoreCase("image/x-fits")) {
      return new com.macfaq.net.www.content.image.x_fits( );
    }
    return null;

  }
}
```

Example 17-11 is a simple program that tests this content handler by loading and displaying a FITS image from a URL. In fact, it can display any image type for which a content handler is installed. However, it does use the `FitsFactory` to recognize FITS images.

Example 17-11. The FITS viewer

```
import java.awt.*;
import javax.swing.*;
import java.awt.image.*;
import java.net.*;
import java.io.*;

public class FitsViewer extends JFrame {

  private URL url;
  private Image theImage;

  public FitsViewer(URL u) {
    super(u.getFile( ));
    this.url = u;
  }

  public void loadImage( ) throws IOException {

    Object content = this.url.getContent( );
    ImageProducer producer;
    try {
      producer = (ImageProducer) content;
    }
    catch (ClassCastException e) {
      throw new IOException("Unexpected type " + content.getClass( ));
    }
    if (producer == null) theImage = null;
    else {
      theImage = this.createImage(producer);
      int width = theImage.getWidth(this);
      int height = theImage.getHeight(this);
      if (width > 0 && height > 0) this.setSize(width, height);
    }

  }
```

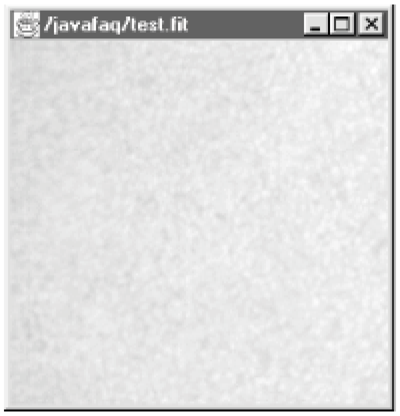
```
    public void paint(Graphics g) {
      if (theImage != null) g.drawImage(theImage, 0, 0, this);
    }

    public static void main(String[] args) {

      URLConnection.setContentHandlerFactory(new FitsFactory( ));
      for (int i = 0; i < args.length; i++) {
        try {
          FitsViewer f = new FitsViewer(new URL(args[i]));
          f.setSize(252, 252);
          f.loadImage( );
          f.show( );
        }
        catch (MalformedURLException ex) {
          System.err.println(args[i] + " is not a URL I recognize.");
        }
        catch (IOException ex) {
          ex.printStackTrace( );
        }
      }
    }
  }
```

The `FitsViewer` program extends `JFrame`. The `main( )` method loops through all the command-line arguments, creating a new window for each one. Then it loads the image into the window and shows it. The `loadImage( )` method actually downloads the requested picture by implicitly using the content handler of Example 17-9 to convert the FITS data into a `java.awt.Image` object stored in the field `theImage`. If the width and the height of the image are available (as they will be for a FITS image using our content handler but maybe not for some other image types that load the image in a separate thread), then the window is resized to the exact size of the image. The `paint( )` method simply draws this image on the screen. Most of the work is done inside the content handler. In fact, this program can actually display images of any type for which a content handler is installed and available. For instance, it works equally well for GIF and JPEG images. Figure 17-2 shows this program displaying a picture of part of solar granulation.

Figure 17-2. The FitsViewer application displaying a FITS image of solar granulation


/javafaq/test.fit