JHOVE Tips for Developers

Gary McGath

Published by Gary McGath

www.garymcgath.com

Other than the license change, it's the same book that I previously sold on Smashwords. It hasn't been updated for the many changes under the Open Preservation Foundation. I got a request for it, so I'm making it available for free. If you have questions, I recommend contacting the Open Preservation Foundation. It's too long since I've worked on the code.

Many thanks to Mark Mandel for proofreading this book. All mistakes are my responsibility, of course.

Table of Contents

Introduction

This book is one part self-promotion, one part practice run for publishing my forthcoming book, *Files that Last*, on Smashwords. It will have a small but hopefully very interested audience: people who use JHOVE in software development work.

There is sample code to go with it on Github. It includes a simple framework class for invoking JHOVE from a Java application and an example of how to extend an output handler. These will be referenced in the upcoming chapters.

It's distributed as a "choose your own price" book, since I wouldn't make any serious money from it no matter what, and I'm guessing I'll actually make more by letting people decide what to pay. If this helps you to accomplish some task, or if you'd just like to support ongoing development of JHOVE, which no one's paying me to do, please consider purchasing this book if you haven't already.

If it isn't January 16, 2013 yet, please look at my Kickstarter project for *Files that Last* and consider supporting it.

Note on code in this book: Smashwords uses software to convert books to many different formats. The style guides impose requirements that conflict with normal code formatting requirements, especially indentation. I've made my best effort to deal with this, but the results may be horrible in some formats, and at best they won't contain hard indents. All code and XML samples can be found on the Github site in the folder codesamples.

Chapter 1: JHOVE basics

(This chapter will appear, with a few changes, in Files that Last.*)*

JHOVE is an open-source application, originally developed at the Harvard University Libraries' Office for Information systems (HUL/OIS, now known as Library Technology Services). It was created in collaboration with JSTOR, a nonprofit organization that maintains an online academic library for multiple institutions. The idea and design were Stephen Abrams'; I wrote most of the code.

JHOVE's purpose is to aid in identifying and validating files for digital preservation. It identifies the type of a file, validates it, and extracts metadata from it. It's used by a number of institutions, often in the process of ingest into a repository.

Currently I'm maintaining it as an open source application independently of Harvard. The source code and latest version can be found on SourceForge. It requires Java 5 or higher and can be run on Windows, Mac OS X, Unix, and Linux systems. This document is based on JHOVE 1.9.

The official documentation is at jhove.sourceforge.net; it's mostly Stephen Abrams' work. It contains essential information; this guide tries to complement it with a more user-friendly approach and more examples.

There are three faces to JHOVE: the library, the command line application, and the GUI application.

Installation

When you download it, you get a TAR or ZIP archive. Expand it by the usual methods and you'll get a directory called `jhove`. On Windows, it's recommended that you put the `jhove` directory under `C:\Program Files`. On Unix-like operating systems you can put it anywhere.

The parts that matter to the user are the top-level directory and the `bin` and `conf` directories. In `bin` you'll find all the JAR files that are needed. If you aren't embedding JHOVE in other software or modifying it, you'll only care about two of these: `JhoveApp.jar` for the command line interface and `JhoveView.jar` for the GUI.

At the top level of the `jhove` directory, there are scripts for Windows and for Unix (including Mac OS X) and Linux. For Windows, the principal script of interest is `jhove.bat`; for the others, it's simply called `jhove`. You'll have to make one edit to the script file for your operating system. Where JHOVE_HOME is defined, you need to set it to the location of your `jhove` directory. In `jhove.bat` on Windows you'll see:

```
SET JHOVE_HOME="C:\Program Files\jhove"
```

If this is where you put the `jhove` directory, you can leave it alone. Otherwise change the path to the one where you actually have it.

On Unix/Linux systems, edit the file `jhove` to fix this line:

```
JHOVE_HOME=[fill in path to jhove directory]
```

to the path for your `jhove` directory, e.g.,

```
JHOVE_HOME=/users/myself/jhove
```

The trickiest thing about setting up JHOVE is often the location of the configuration file. This is found in `jhove/conf/jhove.conf` under your user directory, but where your "user directory" is can vary. On Unix/Linux systems, it's the directory where you find yourself by default when you log in to a shell or enter `cd` with no arguments. On Mac OS X, you can also find it as `/Users/`*`[username]`*. With Windows, it varies from one version to another. Possible locations include:

Windows XP: `C:\Documents and Settings\`*`[username]`*
Windows NT: `C:\WINNT\Profiles\`*`[username]`*
Windows Vista and 7: `C:\Users\`*`[username]`* or perhaps `C:\`*`[username]`*

From version 1.8 on, JHOVE will create a default configuration file if it can't find one. This is actually the better way to do it, since it will save you a step. You have to make one change if you copy the standard file there. Look for a line like this near the top:

```
<jhoveHome>/users/stephen/projects/jhove</jhoveHome>
```

You need to change the path to your JHOVE home directory.

One other change is highly recommended. For the declaration of the TIFF module, add one line to it so that you have the following:

(codesamples/moduledecl.txt)

```
<module>

  <class>edu.harvard.hul.ois.jhove.module.TiffModule</class>
  <param>byteoffset=true</param>

</module>
```

This will be explained more in the discussion of the TIFF module. The rest of the defaults will let you get by, so I'll get into how to do things with JHOVE before coming back to configuration details.

JHOVE looks at a file or directory of files, examining each one using its format modules. You can test a file against just one module or against all the modules specified in the configuration file (which is all of them by default). The file can be on your computer or on the Web. For each file, it will tell you if the file is "well-formed" and "valid." If it's well-formed, it will give you metadata about the file in the format specified by the output handler. The standard version lets you get output with XML, text, and the more specialized audit handler. JhoveView, the GUI version, will show you a result window which you can save using any of the handlers.

Command-line JHOVE

To run the command line version of JHOVE, you'll use the batch file `jhove.bat` on Windows or the shell script `jhove` elsewhere. Both work the same way, except for a few differences required by the operating systems. In both cases you type just `jhove` at the command prompt, followed by the arguments. The simplest thing you can do is to run it with no arguments:

```
jhove
```

You should get back a description of the application and modules. If you get a Java error, the likeliest causes are that the system can't find Java or you didn't set up `JHOVE_HOME` correctly. If it didn't find Java, you need to set up your path correctly for the command line. If you get a Java stack dump, check if your `JHOVE_HOME` directory is correct.

The next thing to try is to run it on a file, e.g.,

```
jhove samplefile.tif
```

This will run all modules on `samplefile.tif` and produce output for the file. You can also run it on a directory:

```
jhove sampledirectory
```

JHOVE will walk through the directory recursively, processing all files.

Let's suppose you know what the intended format is. You can test just for it by selecting the appropriate module:

```
jhove -m TIFF-hul samplefile.tif
```

The advantage of this, aside from speed, is that if the file isn't well-formed TIFF, you'll get error messages from the TIFF module instead of being told it's a well-formed bytestream. The modules to choose from in the standard version are AIFF-hul, ASCII-hul, BYTESTREAM, GIF-hul, HTML-hul, JPEG-hul, JPEG2000-hul, PDF-hul, TIFF-hul, UTF8-hul, WAVE-hul, and XML-hul.

By default the text handler formats the output. If you want XML output, specify

```
jhove -m TIFF-hul -h XML samplefile.tif
```

To make the choice of the text handler explicit, you can specify `-h TEXT`. For the Audit handler if you have a use for it, it's `-h Audit`.

Either way it will write to standard output, normally the console, but you can specify the output file with `-o`:

```
jhove -m TIFF-hul -h XML -o analysis.xml samplefile.tif
```

There's also a capitalized `-H` option to get information *about* the handler. You won't need it very often, but don't confuse the two. You can tell it, for instance:

```
jhove -h XML -H TEXT
```

That will give you information about the text handler, in XML format.

The other parameters are less likely to be useful. I'll just cover them quickly.

```
-c [/config/file/path]
```

Normally your configuration file will be `jhove/conf/jhove.conf` in your home directory, but if it's at another location (perhaps because you have multiple configurations) you can specify it.

```
-b [size]
```

Sets the buffer size; if it's omitted, the JVM default is used. Starting with JHOVE 1.9, the size is set to

1024 if you give a smaller number.

```
-x [fully.qualified.class.name]
```

Sets the XML reader class; the name provided must implement `org.xml.sax.XMLReader`. This affects only the XML module. Older versions of Java came with a feeble XML reader implementation and this option was useful.

```
-t [path]
```

Sets the temporary directory.

```
-l [loglevel]
```

Sets the log level for debugging.

```
-e [encoding]
```

Sets the output encoding. The default is UTF-8.

```
-k
```

Tells JHOVE to calculate CRC32, MD5, and SHA-1 digests (somewhat inaccurately called "checksums"). This slows processing down a lot.

```
-r
```

Show properties as raw numeric values rather than text strings.

```
-s
```

Only check file signatures.

The configuration file

Let's look now at the configuration file, `jhove.conf`. It's an XML file that lets you set various options. You can edit it by hand or through the GUI. Normally you should put it in `jhove/conf/jhove.conf` under your user home directory; that will require the least amount of manual tweaking. Here's what the file looks like out of the box:

(codesamples/configfile.txt)

```
<?xml version="1.0" encoding="UTF-8"?>

<jhoveConfig version="1.0"

  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

  xmlns="http://hul.harvard.edu/ois/xml/ns/jhove/jhoveConfig"

   xsi:schemaLocation="http://hul.harvard.edu/ois/xml/ns/jhove/jhoveConfig

    http://hul.harvard.edu/ois/xml/xsd/jhove/1.4/jhoveConfig.xsd">

  <jhoveHome>/users/stephen/projects/jhove</jhoveHome>
```

```
<defaultEncoding>utf-8</defaultEncoding>

<tempDirectory>/var/tmp</tempDirectory>

<bufferSize>131072</bufferSize>

<module>

  <class>
  edu.harvard.hul.ois.jhove.module.AiffModule
  </class>

</module>

<module>

  <class>
  edu.harvard.hul.ois.jhove.module.WaveModule
  </class>

</module>

<module>

  <class>
  edu.harvard.hul.ois.jhove.module.PdfModule
  </class>

</module>

<module>

  <class>
  edu.harvard.hul.ois.jhove.module.Jpeg2000Module
  </class>

</module>

<module>

  <class>
  edu.harvard.hul.ois.jhove.module.JpegModule
  </class>

</module>

<module>

  <class>
  edu.harvard.hul.ois.jhove.module.GifModule
  </class>

</module>

<module>
```

```
    <class>
    edu.harvard.hul.ois.jhove.module.TiffModule
    </class>

  </module>

  <module>

    <class>
    edu.harvard.hul.ois.jhove.module.XmlModule
    </class>


    <param>
    schema=http://www.example.com/exampleschema.xsd
    </param>

  </module>

  <module>

    <class>
    edu.harvard.hul.ois.jhove.module.HtmlModule
    </class>

  </module>

  <module>

    <class>
    edu.harvard.hul.ois.jhove.module.AsciiModule
    </class>

  </module>

  <module>

    <class>
    edu.harvard.hul.ois.jhove.module.Utf8Module
    </class>

  </module>

</jhoveConfig>
```

As mentioned earlier, if you let JHOVE create `jhove.conf` for you, you don't have to set the home directory here. Otherwise you have to set the path in the `jhoveHome` element to your own JHOVE directory.

Other than that, the likeliest change you'll want to make is to edit the list of modules. If there are file formats you're sure you'll never want to deal with, you can gain some efficiency by removing them (or commenting them out). In particular, the HTML module is slow and not very useful. On the one hand, most HTML files have syntactic errors according to the spec but work anyway. On the other, an HTML file can omit almost every tag that's normally desirable (even `head` and `body`) and still be syntactically well-formed.The JHOVE approach just doesn't mesh well with the HTML reality.

The order of the modules is important in certain cases. XML and HTML need to come before UTF-8 and ASCII, since they're the more specific categories. If you put them in the other order, they'll be identified as the more generic formats. BYTESTREAM is always the implicit last format and isn't mentioned in the configuration file because it can't be removed.

There can be `<outputHandler>` elements if you add custom output handlers. Look at Chapter 3 if you're interested in doing this.

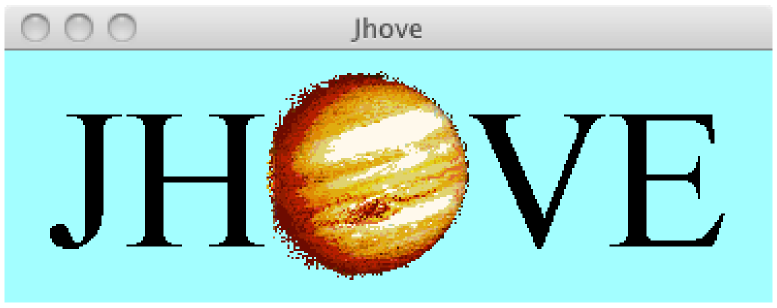
You can set the `<tempDirectory>` element if you're dissatisfied with the default for some reason, and the `<bufferSize>` element to optimize performance. It's simplest to do this with the GUI application rather than worry about the details of the XML schema.

The JHOVE GUI

Another way to run JHOVE is through a graphic user interface. It doesn't give you all the options of the command-line version but may be more convenient. To do this, double-click `bin/JhoveView.jar` or enter

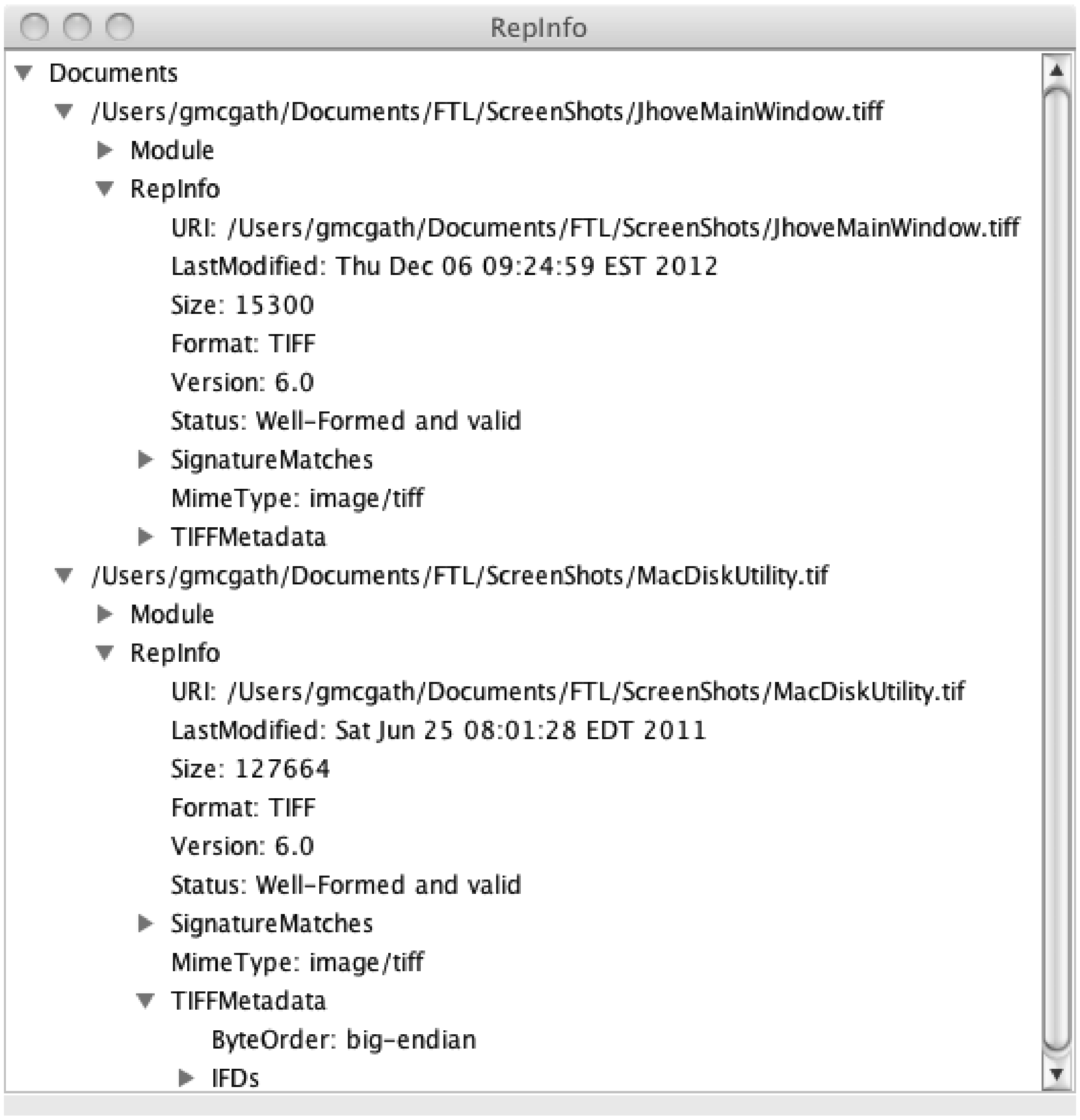
```
jhove -jar bin/JhoveView.jar
```

You should see a window like this:



The configuration file must be at `jhove/conf/jhove.conf` in your home directory. If JHOVE doesn't find one it will create one.

To process a file, select **File > Open...** from the menu. It will analyze the file and put up a window with the information. You can then select **File > Save as...** to save the results as a text or XML file. These files are created with the same handlers used from the command line.

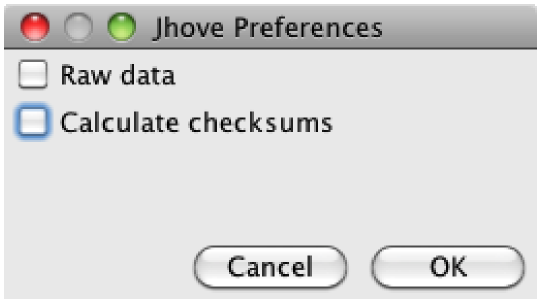


```
▼ Documents
    ▼ /Users/gmcgath/Documents/FTL/ScreenShots/JhoveMainWindow.tiff
        ▶ Module
        ▼ RepInfo
            URI: /Users/gmcgath/Documents/FTL/ScreenShots/JhoveMainWindow.tiff
            LastModified: Thu Dec 06 09:24:59 EST 2012
            Size: 15300
            Format: TIFF
            Version: 6.0
            Status: Well-Formed and valid
          ▶ SignatureMatches
            MimeType: image/tiff
          ▶ TIFFMetadata
    ▼ /Users/gmcgath/Documents/FTL/ScreenShots/MacDiskUtility.tif
        ▶ Module
        ▼ RepInfo
            URI: /Users/gmcgath/Documents/FTL/ScreenShots/MacDiskUtility.tif
            LastModified: Sat Jun 25 08:01:28 EDT 2011
            Size: 127664
            Format: TIFF
            Version: 6.0
            Status: Well-Formed and valid
          ▶ SignatureMatches
            MimeType: image/tiff
          ▼ TIFFMetadata
                ByteOrder: big-endian
              ▶ IFDs
```

You can also drag files to the main window. This lets you analyze more than one file, including entire directories. All the results will appear in one window.

To use just one module, select **Edit > Select module >** ***[module]*** from the main window.

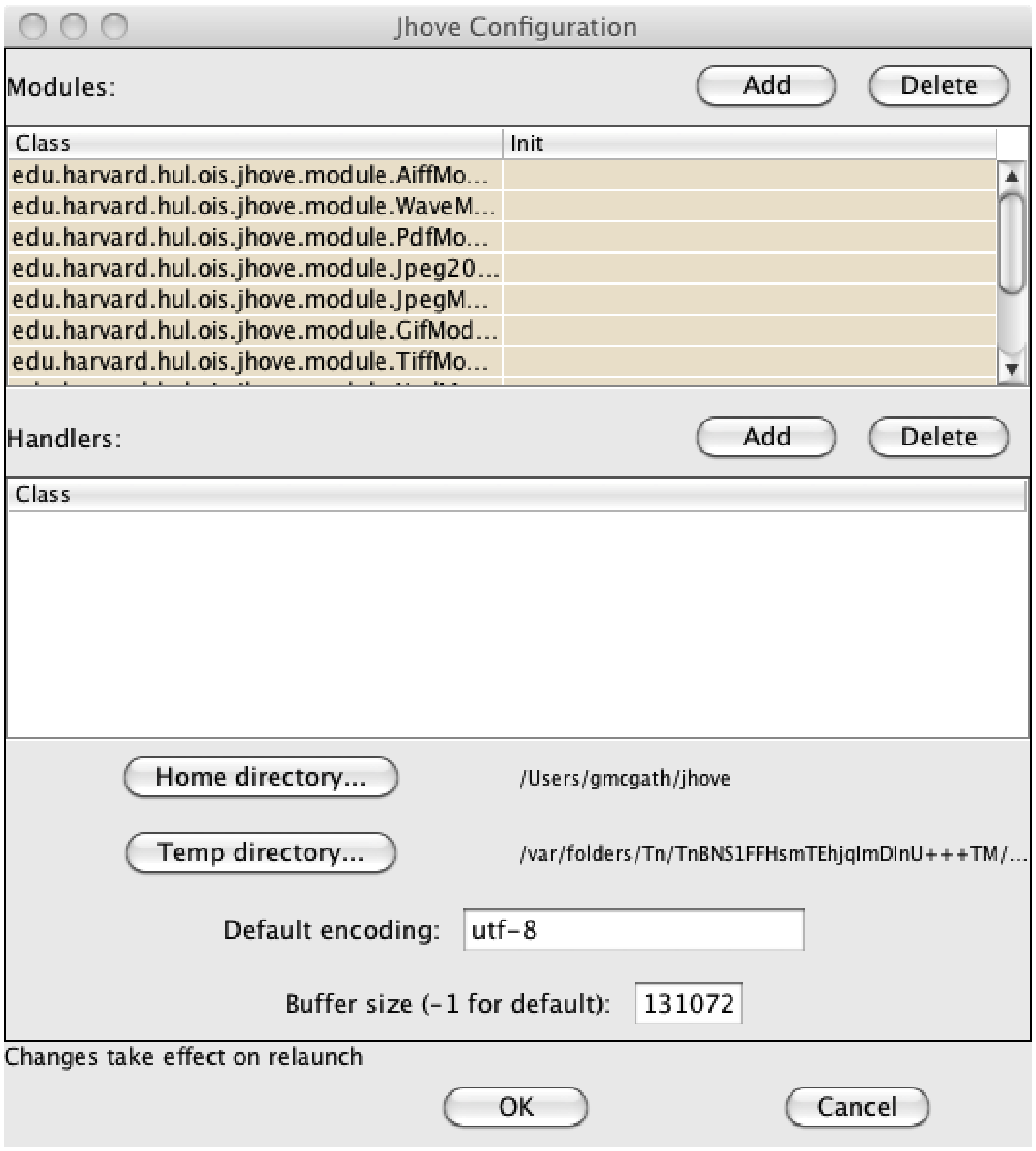
The Preferences window, brought up by **Edit > Preferences...**, has checkboxes for the equivalent of the `-k` and `-r` options in the command line.


Jhove Preferences

- [ ] Raw data
- [ ] Calculate checksums

Cancel OK

Edit > Edit configuration... lets you modify the configuration file. All changes take effect only when you relaunch JHOVE. They also affect the command line version, if it's using the same configuration file. JHOVE 1.8 or earlier has some serious bugs here, but it works starting in 1.9. What you're most likely to do here is remove modules that you don't need.


Jhove Configuration

Modules: Add Delete

Class	Init
edu.harvard.hul.ois.jhove.module.AiffMo...	
edu.harvard.hul.ois.jhove.module.WaveM...	
edu.harvard.hul.ois.jhove.module.PdfMo...	
edu.harvard.hul.ois.jhove.module.Jpeg20...	
edu.harvard.hul.ois.jhove.module.JpegM...	
edu.harvard.hul.ois.jhove.module.GifMod...	
edu.harvard.hul.ois.jhove.module.TiffMo...	

Handlers: Add Delete

Class

Home directory... /Users/gmcgath/jhove

Temp directory... /var/folders/Tn/TnBNS1FFHsmTEhjqlmDlnU+++TM/...

Default encoding: utf-8

Buffer size (-1 for default): 131072

Changes take effect on relaunch

OK Cancel

You can add classes for other modules; they need to be in the classpath to be usable, which means you'll have to launch JhoveView from the command line or a script rather than directly from the JAR file. You can specify the `init` parameter for any modules. However, none of the modules in the default build use it.

To check what version of JHOVE you have, use **Help > About Jhove...**

The library

The JAR files provided in the JHOVE delivery can be used as a library for constructing new applications. This will be covered in the next chapter.

The output

JHOVE identifies files as "well-formed" and "valid" or not. These are terms that come from XML and often are a rougher fit for other formats. With XML, "well-formed" means that a file obeys all the syntactic rules, and "valid" means that it conforms to its schema or DTD declarations. With other formats, JHOVE takes "well-formed" to mean that the structure of the file conforms to the specification, and "valid" to mean that its content does. This is often a hard distinction to make.

The code is nitpicky by design. To be well-formed, a file must follow the format spec exactly. Sometimes there are generally accepted deviations, but JHOVE doesn't accept them by default. There's one case where the rules can be loosened; see the discussion of the TIFF module further on.

Although it's strict with what it checks, JHOVE doesn't analyze encoded data bitstreams, such as image or audio data. This was a decision made at the start of the project; with one manager and one programmer, we just didn't have the resources to properly treat those streams, which often can come in many different encodings within one format.

This means that JHOVE can be overly nitpicky and at the same time miss errors in files. It does what it's designed for. Don't treat it as an oracle, assuming that files it rejects are useless or that ones its accepts are flawless.

What does JHOVE tell you about a file? Lots.

First you'll see which module accepted it, or the last module to process it. If you don't restrict JHOVE to a single module, it will apply all the ones in the configuration file and end with BYTESTREAM, which accepts any readable file. You'll also see the version and date of the module, but ignore it. The original idea was that with every release, the version information in each module would be updated. That just hasn't happened.

Next you'll see the "RepInfo" for the file. How this looks will depend on whether you're using the GUI or using the text or XML handler, but the content will be the same. The main sections you'll see are:

- URI
- LastModified
- Size
- Format
- Status
- SignatureMatches
- MimeType

• Profiles
• Metadata (name varies with the file type)

Most of these should be self-explanatory. SignatureMatches reports any modules that say the file matches an expected “signature,” usually in the first few bytes, even if it isn’t well-formed or valid. For instance, with a TIFF file which has the standard TIFF header, JHOVE will list “TIFF-hul” under “SignatureMatches.” Not all modules do signature checking.

Profiles are subcategories of the format which JHOVE checks for. Examples are PDF/A, GIF 89a, and TIFF/EP. To satisfy a profile, a file must be considered well-formed and valid by JHOVE and pass additional tests for the profile. One file may satisfy several profiles.

The properties reported will be completely different for each module.

The modules

AIFF

AIFF (Audio Interchange File Format) doesn’t get as much use as MP3 and WAVE, but it hangs on. It’s an interchange format intended for long-term storage of audio data, so it falls under JHOVE’s natural coverage. The module recognizes two profiles, which really are subtypes: AIFF and AIFF-C. The AIFF subtype (or if you prefer, “real” AIFF) allows only uncompressed data; AIFF-C allows lossy compressed data. JHOVE checks only the filetype reported in the file and doesn’t check whether appropriate compression algorithms are used.

The module checks the chunk structure but not the data within chunks. This means it won’t catch encoding errors.

Profiles:

AIFF
AIFF-C

Some interesting metadata;

AESAudioMetadata:Face:TimeLine
AESAudioMetadata:Region:NumChannels
AESAudioMetadata:FormatList:FormatRegion:BitDepth
AESAudioMetadata:FormatList:FormatRegion:SampleRate

ASCII

The ASCII module is straightforward. It reports what control characters are used and what line endings are used (CR, LF, CR/LF, but not the rare LF/CR). If more than one kind of line ending occurs in a file it reports all of them.

Profiles:

None

Some interesting metadata:

LineEndings
ControlCharacters

BYTESTREAM

The module of last resort. Everything is a bytestream. However, if the module runs into any I/O errors reading a file, it will report the error rather than saying it's a bytestream.

Profiles:

None

Metadata:

No module-specific metadata

GIF

There are two flavors of GIF: 87a and 89a. The order of the metadata items under Blocks is significant, defining a sequence of operations in the case of animated GIFs.

Profiles:

GIF 87a
GIF 89a

Some interesting metadata:

Blocks:LogicalScreenDescriptor:LogicalScreenWidth
Blocks:LogicalScreenDescriptor:LogicalScreenHeight
ApplicationExtension
GraphicControlExtension:TransparencyFlag
GraphicControlExtension:DelayTime
ImageDescriptor:NisoImageMetadata:ImageWidth
ImageDescriptor:NisoImageMetadata:ImageLength
ImageDescriptor:NisoImageMetadata:BitsPerSample

HTML

HTML is one of the messiest formats there is, and it goes against the whole JHOVE approach. Read the specification, and you discover that virtually every tag is optional. A text file that has a tag somewhere in the middle may well meet the definition of an HTML document. Unlike the other modules, this one relies heavily on heuristics. It recognizes XHTML and hands it off to the XML module. To accept a document as HTML, it requires at least one HTML, HEAD, BODY, or TITLE tag. The module knows nothing about HTML5.

If you remove this module from `jhove.conf`, you'll get better performance; there are better ways to validate HTML.

Profiles:

Strict
Frameset
Transitional

Some interesting metadata:

PrimaryLanguage
Title

JPEG

The JPEG module analyzes what are commonly called JPEG files. These are actually JFIF (JPEG Files Interchange Format) files or less often the similar SPIFF (Still Picture File Interchange Format). “JPEG” properly refers to the container and encoding, not the file format. The module recognizes the Exif profile, which means that the image has Exif metadata in the form of a TIFF IFD, and the JPEG-LS profile, which indicates the use of lossless encoding. The JPEG module invokes the TIFF module to get Exif metadata.

The module checks the file structure but not the encoded image data. It looks for XMP metadata but just reports it as raw XML.

Profiles:

JFIF
SPIFF
Exif
JPEG-L

Some interesting metadata:

CompressionType
Images:Image:NisoImageMetadata
Images:Image:Exif
Images:Image:XMP

JPEG2000

Like JPEG, JPEG2000 proper defines a container and an encoding rather than a file format. The two primary JPEG2000 file formats are JP2 and JPX. The JPEG2000 module recognizes both of these. The module checks the file structure but not the encoded image data.

Profiles:

JP2
JPX

Some interesting metadata:

Compatibility

ColorSpecs
Codestreams:Codestream:ImageAndTileSize:XSize
Codestreams:Codestream:ImageAndTileSize:YSize
Codestreams:Codestream:NisoImageMetadata

PDF

The PDF module covers versions of PDF through 1.6; it hasn't been updated for PDF 1.7 (ISO 32000), though most files created under PDF 1.7 will pass muster. It recognizes lots of profiles, including the preservation-friendly PDF/A-1 (but not A-2 or A-3). The PDF/X profile, designed for graphics interchange, is recognized through version 3 but not version 4. The module doesn't validate data within content streams (including operators) or encrypted data.

Profiles:

ISO PDF/A-1, Level A
ISO PDF/A-1, Level B
Linearized PDF
Tagged PDF
ISO PDF/X-1
ISO PDF/X-1a
ISO PDF/X-2
ISO PDF/X-3

Some interesting metadata:

DocumentCatalog:PageLayout
DocumentCatalog:PageMode
Fonts
Info:Title
Info:Author
Info:CreationDate
XMP

TIFF

The TIFF module covers the TIFF specifications through 6.0, plus various technical notes updating the format since 1992. As mentioned in the Installation section of Chapter 1, the configuration file lets you excuse a deviation from the default format which is widely accepted, by setting the parameter `byteoffset=true` (exactly those characters, all lower case with no spaces). The TIFF specification says that all IFD (image file directory) value offsets must be even numbers, so that all values are aligned on word (16-bit) boundaries. Many creators of TIFF files ignore this requirement, and just about all TIFF readers accept files that do. Unless you have reason to require literal adherence to the spec, make the change to the configuration file mentioned above. You can copy it from `jhove/conf/jhove-byteoffset=true.conf` in the downloaded distribution.

The TIFF module recognizes a vast collection of profiles. Most of them fall under the TIFF/IT set of standards. TIFF/IT actually defines sets of files: Final Page or FP, Contone or CT, Linework or LW, and sometimes High Resolution Contone (HC), Binary Linework (BL), Binary Picture (BP), and

Monochrome Picture (MP). Each of these has revisions, so that adds up to a lot of profiles. GeoTIFF allows georeferencing data in a standard way, and TIFF/EP is a photographic image format based in part on Exif. The DLF (Digital Library Federation) benchmarks can be considered obsolete.

A TIFF file may have multiple IFDs, not all of which necessarily correspond to images. Useful metadata can be found in the IFDs of type GlobalParameterIFD and Exif.

Profiles:

Baseline bilevel (Class B)
Baseline grayscale (Class G)
Baseline palette-color (Class P)
Baseline RGB (Class R)
Extension YCbCr (Class Y)
TIFF/IT-BL (ISO 12639:1998)
TIFF/IT-BL/P1 (ISO 12639:1998)
TIFF/IT-BP (ISO 12639:1998)
TIFF/IT-BP/P1 (ISO 12639:1998)
TIFF/IT-BP/P2 (ISO 12639:1998)
TIFF/IT-CT (ISO 12639:1998)
TIFF/IT-CT/P1 (ISO 12639:1998)
TIFF/IT-CT/P2 (ISO 12639:2003)
TIFF/IT-FP
TIFF/IT-FP/P1 (ISO 12639:1998)
TIFF/IT-FP/P2 (ISO 12639:2003)
TIFF/IT-HC (ISO 12639:1998)
TIFF/IT-HC/P1 (ISO 12639:1998)
TIFF/IT-HC/P2 (ISO 12639:2003)
TIFF/IT-LW (ISO 12639:1998)
TIFF/IT-LW/P1 (ISO 12639:1998)
TIFF/IT-LW/P2 (ISO 12639:2003)
TIFF/IT-MP (ISO 12639:1998)
TIFF/IT-MP/P1 (ISO 12639:1998)
TIFF/IT-MP/P2 (ISO 12639:2003)
TIFF/IT-SD (ISO 12639:2003)
TIFF/IT-SD/P2 (ISO 12639:2003)
TIFF/EP (ISO 12234-2:2001)
Baseline GeoTIFF 1.0
DLF Benchmark for Faithful Digital Reproductions of Monographs and Serials: black and white
DLF Benchmark for Faithful Digital Reproductions of Monographs and Serials: grayscale and white
DLF Benchmark for Faithful Digital Reproductions of Monographs and Serials: color
RFC 1314
TIFF-FX (Profile S)
TIFF-FX (Profile F)
TIFF-FX (Profile J)
TIFF-FX (Profile L)
TIFF-FX (Profile C)
TIFF-FX (Profile M)
Exif 2.0
Exif 2.1 (JEIDA-49-1998)

Exif 2.2 (JEITA CP-3451)
DNG 1.0.0.0 (September 2004)

Some interesting metadata:

IFDs:IFD:Entries:NisoImageMetadata:CompressionScheme
IFDs:IFD:Entries:NisoImageMetadata:ImageWidth
IFDs:IFD:Entries:NisoImageMetadata:ImageLength
IFDs:IFD:Entries:NisoImageMetadata:BitsPerSample
IFDs:IFD:Entries:NisoImageMetadata:SamplesPerPixel

Interesting metadata specific to an IFD of type `Exif`:

IFDs:IFD:Entries:ExposureTime
IFDs:IFD:Entries:ShutterSpeedValue
IFDs:IFD:Entries:ApertureValue
IFDs:IFD:Entries:MeteringMode
IFDs:IFD:Entries:Flash

UTF-8

The UTF-8 module determines if a file is good UTF-8 and what code blocks it uses. This could be a clue to what language the document is in. The "Characters" metadata item reports the character count, which can be different from the byte count. The presence of a BOM is reported under "Messages," not under UTF-8 metadata. There are no profiles.

Some interesting metadata:

Characters
UnicodeBlocks
LineEndings

WAVE

The WAVE audio format is short on published specifications, which made writing the module interesting. An open-ended set of codecs is allowed, though certain ones are common. The module doesn't validate any of the encodings.

Profiles:

PCMWAVEFORMAT
WAVEFORMATEX
WAVEFORMATEXTENSIBLE
Broadcast Wave Version 0
Broadcast Wave Version 1

Some interesting metadata:

AESAudioMetadata:Face:Timeline:Start
AESAudioMetadata:Face:Region:TimeRange

AESAudioMetadata:CompressionCode
AESAudioMetadata:AverageBytesPerSecond

XML

XML is a close cousin of HTML; yet while the HTML module finds itself with a nearly hopeless task analyzing an ill-defined format, the XML module has the cleanest job of any of the modules. There's no ambiguity about the definitions of "well-formed" and "valid." Most of the work is done by the XML parser.

There is one issue, though. Determining if a file is valid can require reading its schemas. This usually means going out to the Internet to retrieve them, and that can really slow processing down or even make it hang if the server is having problems, or if it slows down your access because you're hitting it too hard. JHOVE 1.8 introduces a technique to avoid this problem with commonly-used schemas. You can create local copies of schemas and declare them in jhove.conf. This might look something like this:

(codesamples/localschemaref.txt)

```
<module>

  <class>edu.harvard.hul.ois.jhove.module.XmlModule</class>

  <param>

    schema=http://www.example.com/schema;

    /home/schemas/exampleschema.xsd

  </param>

</module>
```

If the parser is incapable of validation, the module will report that validity is undetermined.

The module recognizes two other parameters:

s	Require an XML declaration
b	The rest of the parameter is a base URL for DTDs. There is no equal sign or any other separator.

The module has no profiles.

Some interesting metadata:

Schemas:Schema
Namespaces:Namespace

Chapter 2: The JHOVE API

Running JHOVE under the control of a custom Java application isn't hard. The application needs to use `JhoveApp.jar`, which is found in the `bin` directory. The central class is `edu.harvard.hul.ois.jhove.JhoveBase`.

`JhoveBase` depends on a configuration file, as discussed in the last chapter, so the first step is to set that up. There's also an option for selecting a SAX class, but that's mostly an issue for older versions of Java, which had a weak default parser. I'll assume you can use the default. Then your initialization code will begin like this:

(codesamples/jhovebase1.txt)

```
JhoveBase jb = new JhoveBase ();

jb.init (configFilePath, null);
```

Next you need to set some characteristics of the `JhoveBase` object. That will look something like this:

(codesamples/jhovebase2.txt)

```
jb.setEncoding ("UTF-8");

jb.setTempDirectory ("/user/me/temp1");

jb.setBufferSize (131072);

jb.setChecksumFlag (false);

jb.setShowRawFlag (false);

jb.setSignatureFlag (false);
```

With these settings, the default output encoding is UTF-8, checksums won't be calculated (a significant time saver if they aren't needed), descriptive strings will be used where possible instead of raw numbers, and signatures-only checking (which is rarely used) is turned off.

`JhoveBase` isn't re-entrant. If you're running with multiple threads, each thread needs to have its own `JhoveBase` object.

You also need an `edu.harvard.hul.ois.jhove.App` object. This just holds some fixed information about your application, so there only needs to be one instance.

(codesamples/app.txt)

```
String appName = "My JHOVE wrapper";

String version = "1.0";

int[] date = { 2002, 12, 31};
// year, month, day

String usage =
"One-line explanation of how to use it";
```

```
String rights = "Copyright 2012 by me";

App app =
new App (appName, version, date, usage, rights);
```

If the application isn't for publication, it doesn't matter very much how you set these.

You can either select a module to process files or let JHOVE apply all the modules in the configuration file. To use them all, it's simple:

```
Module module = null;
```

Otherwise select a module of the desired type:

```
Module module = new AsciiModule();
```

You need to select an output handler, which can't be null. If you want to do something unusual with the output, you'll have to either post-process it or write a custom handler.

```
OutputHandler handler = new XMLHandler();
```

You can provide the path of an output file. If you make it `null`, JHOVE will write to `System.out`.

```
String outFile = "output/outputfile.xml";
```

Next, you need to specify what you're processing. The string can be the path of a file or directory or a URI (which had better be a URL to do anything useful). HTTPS URLs will be processed without any authentication. If the string is a directory path, JHOVE will walk down through all subdirectories and process all the files in them.

```
String f = "/users/me/filestoprocess";
```

Finally you call the dispatcher:

```
jb.dispatch (app, module, null, handler, outFile, f);
```

It's a good idea to wrap the call in a `try/catch` loop in case any bugs throw uncaught exceptions.

Alternatively, you can get more control by calling `process` rather than `dispatch`. This way you handle one file at a time and you aren't forced to send output to a file, but you do more initialization. Here's a minimal example:

(codesamples/process.txt)

```
PrintWriter writer = ...;

handler.setWriter (writer);

handler.setBase (jb);

Module module = ...;

module.init("");

module.setDefaultParams(new ArrayList<String>());
```

```
jb.process (app, module, handler, dirFileOrUri);
```

The same `Module` and `OutputHandler` instances can be used for any number of calls to `dispatch`.

Chapter 3: Custom output

JHOVE's output can be customized by writing a new output handler or subclassing an existing one. Any handler has to implement the interface `edu.harvard.hul.ois.jhove.OutputHandler`, and it can almost always subclass `edu.harvard.hul.ois.jhove.HandlerBase`. An output handler doesn't have to write to a file; it can process the information which the module provides in just about any way, such as transforming it so that the application can continue using it.

The module creates a `RepInfo` object, which the handler processes. Let's look first at the property map, which is returned by

```
public Map<String, Property> getProperty ();
```

In essence a `Property` is a name-value pairing, but it can have different kinds of values, and the design is less clean than it might be today because it was written for Java 1.4, which predated generics and enumerations. I've added generics to large parts of the current version, but JHOVE could have been written a bit differently if they'd been available from the beginning.

The two most important functions of `Property` are:

```
public String getName ();
```

```
public Object getValue ();
```

To find out what kind of object a property's value is, you call its `getArity()` function:

```
public PropertyArity getArity ();
```

Today that would be implemented as an enumeration. It's a class which is implemented as several static instances, called `ARRAY`, `LIST`, `MAP`, `SCALAR`, and `SET`.

To find out the type of individual data items in a property, you call `getType()`.

```
public PropertyType getType ();
```

This likewise has a set of static instances, called `BOOLEAN`, `BYTE`, `CHARACTER`, `DATE`, `DOUBLE`, `FLOAT`, `INTEGER`, `LONG`, `OBJECT`, `AESAUDIOMETADATA`, `NISOIMAGEMETADATA`, `TEXTMDMETADATA`, `PROPERTY`, `SHORT`, `STRING`, and `RATIONAL`. If the Arity is `SCALAR`, the Type represents the type of the value. Otherwise it represents the type of an element of the collection which is the value. For instance, if the Arity is `LIST` and the Type is `DATE`, then the value will be a `List<Date>`.

The `PROPERTY` type is especially interesting because it allows constructing hierarchies of properties. JHOVE makes extensive use of this.

Now we can get to writing the output handler itself. I'll assume you subclass `ModuleBase`, which does a lot of the setup work for you. Here are the functions you may need to implement:

```
public void init (String init) throws Exception;
```

This does one-time initialization. The format and use, if any, of the parameter is up to you.

```
public void reset ();
```

This does any initialization which is needed before each call to the handler.

```
public boolean okToProcess (String filepath);
```

This is called for each file before it's processed. By returning false, the handler can veto processing of any file or URI.

```
public void analyze (RepInfo info);
```

None of the handlers call this, but it can be useful if you want to subclass an output handler. It's called before any output is done and can be used to pre-process the `RepInfo`. You could use this for things like deleting uninteresting metadata or collecting statistics.

```
public void endDirectory ();
```

If you need to do anything special when you're finished with a directory, you can put it here. Normally no action is needed.

```
public void show (RepInfo info);
```

This is called after everything else is set up to output the `RepInfo` object. This is normally where most of your work will be, unless you're subclassing an existing handler and don't have to change this.

```
public void show (Module module);
```

This outputs information about a Module.

```
public void show (OutputHandler handler);
```

This outputs information about an OutputHandler, possibly itself, possibly another. For other OutputHandlers to produce interesting information about your handler, you have to initialize certain variables in your constructor. Look at the constructors of the existing handlers for examples.

```
public void show ();
```

This "outputs minimal information about the application." Existing handlers output nothing.

Processing RepInfo

Let's get back to the `RepInfo` object and look at more of its functions, which an output handler may need to call.

```
public String getFormat ();
```

This returns the name of the format as given by the matching module.

```
public Date getLastModified ();
```

This returns the modification date of the object.

```
public List<Message> getMessage ();
```

This returns a list of message objects, which may be of type `InfoMessage` or `ErrorMessage`, generated by the module.

```
public String getMimeType ();
```

This returns the MIME type assigned by the module.

```
public List<String> getProfile ();
```

This returns the list of profile names which the module matches with the document.

```
public Property getProperty (String name);
```

This returns a named top-level property.

```
public Property getByName (String name);
```

This returns a named property anywhere in the hierarchy. If there's more than one property with the same name, which one will be returned isn't specified.

```
public long getSize ();
```

This returns the size of the document in bytes.

```
public String getUri ();
```

This returns the URI of the document.

```
public boolean getURLFlag ();
```

This returns `true` if the document was obtained from a URL.

```
public int getWellFormed ();
```

This returns `RepInfo.TRUE`, `RepInfo.FALSE`, or `RepInfo.UNDETERMINED`. JHOVE doesn't currently have any cases where the value will be `UNDETERMINED`.

```
public int getValid ();
```

This returns `RepInfo.TRUE`, `RepInfo.FALSE`, or `RepInfo.UNDETERMINED`. With the standard modules, the only case in which `UNDETERMINED` is returned is if the XML parser is incapable of validation.

```
public String getVersion ();
```

This returns the version of the format which the document satisfies (e.g., "3.0").

```
public String getNote ();
```

Returns the note supplied by the module. May be null.

```
public List<String> getSigMatch ();
```

Returns the list of *all* signature strings found by any module to match the document. May be empty. This is the one case where modules other than the last one to process the document contribute to `RepInfo`.

An example: FilteredXmlHandler

If you haven't already, grab the sample code from Github. It includes the class `FilteredXmlHandler`, which extends `XmlHandler` with code to prune the properties output.

The constructor calls the superclass constructor, providing its own identification information.

This handler's contribution consists of working its way down the properties tree and keeping only properties whose name is on a list of "significant" properties. If a property has a significant name, all its descendant properties are retained. The function `addSignificantProperty` lets you specify as many significant names as you want.

FilteredXmlHandler uses the `analyze` function to pre-process the `RepInfo` object. It uses `PropertyPruner` to remove unwanted properties. The pruning is done in a separate class because exactly the same trick could be done in a subclass of `TextHandler` or in the GUI version of JHOVE. `PropertyPruner` calls `RepInfo.getProperty` to get the top-level property map and works recursively through any descendant properties.

It doesn't have to override any other classes, since it otherwise acts just like `XmlHandler`.

Chapter 4: Modules

JHOVE modules check files against specific formats to determine whether a file is well-formed or not and valid or not, and to extract metadata.

There is a guide to writing a module on the JHOVE website. This chapter will refer to it while looking at existing modules for examples.

All modules implement the `Module` interface. All existing modules subclass `ModuleBase`, and it's hard to think of a reason not to. There are two broad kinds of modules, those which use random access and those which go through a file serially. You may want to refer to `TiffModule` as a random access example and `AsciiModule` as a serial access example; I'll be referring mostly to them in the following.

Each module's constructor should call its `super` constructor; there's a long list of arguments, most of which are fixed descriptive strings; the last one is `true` if the module will use random access and `false` if it will use serial access. The first line of the constructor will look something like this:

```
super (NAME, RELEASE, DATE, FORMAT,
   COVERAGE, MIMETYPE, WELLFORMED,
   VALIDITY, REPINFO, NOTE, RIGHTS, isRandomAccess);
```

If you look at the modules, you'll see that the constructor can add a lot more information about the module. There really isn't much reason to, though.

You'll need to subclass one of two `parse` functions, depending on whether your module is random access. If it is, you'll need:

```
public final void parse (RandomAccessFile raf, RepInfo info)
   throws IOException;
```

If it uses serial access, you'll need:

```
public final int parse
   (InputStream stream, RepInfo info, int parseIndex)
   throws IOException;
```

The `parseIndex` argument is needed because some modules need to run through the file more than once. It's called the first time with a value of 0. If it returns 0, the parsing is done; otherwise it will be called again with the returned value as the new `parseIndex`. `XmlModule` uses this feature.

The module parameters may affect the parsing operation. If so, check `_defaultParams` in the `parse` function. The values to look for in the parameters are whatever you decide they should be. `AsciiModule` checks if it should produce TextMD metadata. The parameter it looks for is "withtextmd=true" — exactly those characters, without spaces or any other alteration.

To read the file efficiently and have access to useful functions, most of the serial access modules call `ModuleBase.getBufferedDataStream()` to create a DataInputStream on top of a BufferedInputStream. They store this in an instance variable, since the Module isn't intended to be thread-safe.

The central job of `AsciiModule` is to check whether a file is ASCII. The code that does it is this:

(codesamples/asciicheck.txt)

```
int ch = readUnsignedByte (_dstream, this);

/* Only byte values 0x00 through 0x7f are valid. */

if (ch > 0x7f) {

  ErrorMessage error =

    new ErrorMessage ("Invalid character",

      "Character = " + ((char) ch) +

      " (0x" + Integer.toHexString (ch) +

      ")", _nByte - 1);

  info.setMessage (error);

  info.setWellFormed (RepInfo.FALSE);

  return 0;
```

This bit of code shows what to do when a file is found not to be well-formed. The variable `info` is the `RepInfo` object. The parser calls `RepInfo.setWellFormed(int)` to mark its status. An int argument is used rather than a boolean because the status can be `UNDETERMINED` as well as `TRUE` or `FALSE`. When a document is marked as not well-formed or not valid, at least one error message should be added to `RepInfo`, and if possible the message should show the file offset at which the error was detected (the second argument to `ErrorMessage`). To mark it as invalid rather than not well-formed, you would call `RepInfo.setValid(RepInfo.FALSE)`. In the present case, there are no ASCII files which are well-formed but not valid.

Notice that `parse` returns as soon as it detects an error. This is typical of JHOVE modules and means they generally report only the first error found. Nothing actually requires this. With the ASCII module, it would be pretty simple to have it count up all the defective characters it finds; with some other modules, the parser would need better recovery techniques to keep going. If you want to make your module plow through as many errors as it can, your users will be happier for it.

In addition to checking for well-formedness, `AsciiModule` gathers various statistics. For example, it checks which control characters are used and puts them into a map, which is reported as a property. Properties are accumulated into the List `metadataList`, which is then added to `RepInfo` as the top-level Property named `ASCIIMetadata`. It's added only if it's not empty; there should never be empty property lists, maps, or sets under `RepInfo`.

Modules have a `checkSignatures` function. If you're going to release a module publicly, you should put something here that checks the beginning of the file for probable conformity to the format. The simplest way to do this is to create a list of `Signature` objects in the constructor. The `ModuleBase` constructor initializes `_signature` for this purpose. The `AiffModule` constructor provides an example of creating signatures:

(codesamples/createsigs.txt)

```
Signature sig = new ExternalSignature ("AIFF",

  SignatureType.FILETYPE,
  SignatureUseType.OPTIONAL);
_signature.add (sig);
sig = new ExternalSignature ("AIFC",
  SignatureType.FILETYPE,
  SignatureUseType.OPTIONAL);
_signature.add (sig);
sig = new ExternalSignature (".aif",
  SignatureType.EXTENSION,
  SignatureUseType.OPTIONAL);
_signature.add (sig);
sig = new ExternalSignature (".aifc",
  SignatureType.EXTENSION,
  SignatureUseType.OPTIONAL,
  "For AIFF-C profile");
_signature.add (sig);

sig = new InternalSignature ("FORM",
  SignatureType.MAGIC,
  SignatureUseType.MANDATORY, 0);
_signature.add (sig);

sig = new InternalSignature ("AIFF",
  SignatureType.MAGIC,
  SignatureUseType.OPTIONAL, 8,
  "For AIFF profile");
_signature.add (sig);

sig = new InternalSignature ("AIFC",
  SignatureType.MAGIC,
  SignatureUseType.OPTIONAL, 0,
  "For AIFF-C profile");
```

```
_signature.add (sig);
```

When there’s a matched signature for a document, you call `RepInfo.setSigMatch(module_name)`. This call has the peculiar feature that its value is retained across all the modules `JhoveBase` checks a given document against, so that a list of matching Modules is accumulated and reported. This can be useful when JHOVE reports only that a document is a bytestream; it may report the signature of the format that the file was intended to be, though JHOVE considers it defective.

Many modules do profile checking. This can get complicated, as the parser has to keep track of whether a file has all required characteristics and doesn’t have any disqualifying ones, and some conditions are context-dependent. `TiffModule` keeps track of a lot of profiles by building a list with `TiffProfile.buildProfileList()` and having each profile check each IFD after it’s parsed. Some of the profiles need special-case treatment. Providing good support for profile checking is one of the weak aspects of the JHOVE architecture, and there can be a variety of solutions. If a document doesn’t satisfy a profile, it simply isn’t listed as having that profile. There are no messages to indicate why a document doesn’t fit a profile.

If a module finds a document well-formed, it should fill in all the applicable fields of the `RepInfo` object. The setter functions correspond to the getters discussed in Chapter 3.

###