Preface

JHOVE2 is a Java framework and application for next-generation format-aware characterization. Characterization is the process of examining a formatted digital source unit and automatically extracting or deriving representation information about that source unit that is indicative of its significant nature and useful for purposes of classification, analysis, and use. For more information, visit http://jhove2.org.

The JHOVE2 Programmer’s Guide is a conceptual introduction to the characterization process and a guide to extending that framework, with a particular emphasis on Format Modules. The reader of this manual should already be familiar with the JHOVE2 User’s Guide.

Note: To simplify discussion, class names are usually given without their package names. An index of class packages is provided at the end of the document. In the PDF version, all class names are clickable links to entries in the package index.

Note: To avoid repetition, names of files are generally specified with POSIX file separators (/). Windows file separators (\) are used only when the file is Windows-specific or different under Windows.

Note: In the PDF version of this document, all references to sections, figures, tables and page numbers are clickable links.

Acknowledgments

The JHOVE2 project is funded by the Library of Congress as part of its National Digital Information Infrastructure Preservation Program (NDIIPP).

Version History

<table>
<thead>
<tr>
<th>Version</th>
<th>Date</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0.1</td>
<td>December 22, 2010</td>
<td>Final Draft</td>
</tr>
<tr>
<td>2.0.0</td>
<td>March 22, 2011</td>
<td>Memory Management Information, other edits</td>
</tr>
</tbody>
</table>

JHOVE2: Next-Generation Architecture for Format-Aware Characterization
# Contents

Preface ........................................................................................................................................3
Acknowledgments ......................................................................................................................3
Version History ..........................................................................................................................3
Contents ......................................................................................................................................4
Figures .......................................................................................................................................5
Tables ........................................................................................................................................5
Understanding Characterization ...............................................................................................6
   Source Unit Concepts ..............................................................................................................6
   The Characterization Process ..................................................................................................6
      A Standard Implementation: Jhove2 .....................................................................................6
      Characterization Example: an ASCII File ..........................................................................9
      Characterization Example: Nested Containers ...............................................................12
Reportables and Modules ...........................................................................................................16
   Reportables in Java Code .........................................................................................................16
      Zero-Argument Constructor ...............................................................................................16
      Defining a Property ............................................................................................................16
      Inherited Properties ...........................................................................................................18
   Spring Configuration of Reportables ....................................................................................18
Format Modules, Format Profiles, and Formats ........................................................................22
Creating a Format Module Class ...............................................................................................22
   Define the Class ....................................................................................................................23
   Define Fields ........................................................................................................................23
   Implement Properties ............................................................................................................24
   Implement Constructors .......................................................................................................24
   Implement a Parser ...............................................................................................................24
   Implement Validator .............................................................................................................25
   Enforce Persistence Requirements ......................................................................................26
   Parsing Aggregate Source Units .........................................................................................26
Creating a Format Profile Class ................................................................................................27
Using Spring to Instantiate Format Modules, Format Profiles, and Formats ..............................28
   Define the Beans for the Format Module and Format Profile ..............................................28
   Define the Beans for Format Objects ...................................................................................30
Where to Find More Information ......................................................... 34
A Format Module Example .................................................................. 35
Package Index .................................................................................... 39

Figures

Figure 1: jhove2 Command Line Application ........................................ 7
Figure 2: Characterization Data for ASCII File ....................................... 12
Figure 3: Directory Characterization Example Data before Aggregation ........ 14
Figure 4: Final Form of Directory Characterization Example ...................... 15
Figure 5: Property Definition with Annotation and Getter ......................... 17
Figure 6: Reportable Property Example with References .......................... 17
Figure 7: beans Stub for XML Configuration file .................................... 19
Figure 8: Simple Bean Defined by Constructor Argument .......................... 20
Figure 9: Bean with Properties .............................................................. 20
Figure 10: List Property ....................................................................... 20
Figure 11: Bean References .................................................................... 20
Figure 12: UTF8 Module Class Declaration ............................................ 23
Figure 13: Format Module Standard Constants ....................................... 23
Figure 14: Variable Field to Store numCharacters Property ...................... 23
Figure 15: Coverage Property Value ..................................................... 23
Figure 16: UTF8Module Constructors .................................................... 24
Figure 17: Beginning of UTF8Module.parse() ......................................... 25
Figure 18: Minimal Bean for UTF8Module ............................................. 29
Figure 19: More Complete Bean for UTF8Module ................................... 29
Figure 20: Agent Bean ......................................................................... 29
Figure 21: Bean for ASCIIProfile ......................................................... 29
Figure 22: Minimal UTF-8 Format Bean with I8R Beans ......................... 31
Figure 23: Complete Format Bean for UTF-8 ........................................ 32
Figure 24: Bean for ASCII Format ......................................................... 33

Tables

Table 1: Selected I8R Values for UTF-8 Files ......................................... 31
Understanding Characterization

Source Unit Concepts

A source unit is any collection of data that JHOVE2 can characterize as a single entity. Among other things, a source unit can be a file, a collection of files, or a stream of bytes within a file.

During the characterization process, all the information relating to a source unit is encapsulated in a Source object.

It’s convenient to organize source units into the following taxonomy:

- **Unitary source units.** Simple source units that JHOVE2 only knows how to characterize as a unit, such as a simple text file.

- **Aggregate source units.** Source units that contain other source units, known as child source units. JHOVE2 can characterize the aggregate as a single entity in addition to characterizing the child source units individually.
  - **Containers.** These are entities such as file system directories and archive files that hold collections of source units that may or may not be related.
  - **Wrappers.** These are entities such as TIFF files that bundle together source units for a specific purpose.
  - **File sets.** The user-specified source units specified on the jhove2 command line. Only defined when multiple source units are specified on a single command line.
  - **Clumps.** Sets of source units that JHOVE2 knows how to characterize as a unit. These include GIS shapefiles, which consist of at least 3 files (with .shp, .shx, and .dbf extensions) possibly accompanied by other files.

Note: Clumps are automatically identified by the aggregation module. The user does not need to identify clumps or separate their constituents from unrelated files.

The Characterization Process

**A Standard Implementation: jhove2**

This section describes the characterization process as it is implemented in the JHOVE2CommandLine command line application. The characterization process is highly configurable, so JHOVE2CommandLine
serves as a prototype for other characterization applications and for JHOVE2 integration in custom workflows.

The command line application is driven by a script that invokes `JHOVE2CommandLine.main()`. This main method implements the following steps:

![Figure 1: jhove2 Command Line Application](image)

**Figure 1: jhove2 Command Line Application**

<table>
<thead>
<tr>
<th>Command Line</th>
<th>Characterize</th>
<th>Parse and Validate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parse command line</td>
<td>Characterize()</td>
<td>Parse and Validate</td>
</tr>
<tr>
<td>Create initial Source object</td>
<td>Identification</td>
<td>Parse</td>
</tr>
<tr>
<td>Invoke Characterize()</td>
<td>Dispatch</td>
<td>Validate</td>
</tr>
<tr>
<td>Display</td>
<td>Digest</td>
<td></td>
</tr>
</tbody>
</table>
1. The command line arguments are parsed to set options and obtain a set of document paths. Each path can be a URL for a remote resource, a local file system file, or a local directory.

2. An initial `Source` object is created to encapsulate all information for the specified documents. If a single path is specified, a single `URLSource`, `FileSource`, or `DirectorySource` object is created. Otherwise, the specified paths are considered a single file set, and a `FileSetSource` object is created, with child source objects for each path.

3. The initial `Source` object is passed to `JHOVE2.characterize()`. This method implements the following sequence:

   a. Identification: a set of presumptive formats are determined. This step is managed by `IdentifierCommand`.

   b. Dispatch: the set of format `modules` needed to parse and validate the formats is determined. No format module is applied to a particular source unit more than once, even if identification returns more than one presumptive format that can be parsed and validated by that format module. For each format module, the following steps occur:

      i. The format module's `parse()` method is invoked to identify the structure and contents of the source unit.

         If the source unit is an aggregate, the parser is responsible for initiating the characterization of the child source units:

         - In containers, wrappers, and file sets, the child source units are not identified until the aggregate is parsed. The parser creates each child's `Source` object and passes it to a recursive invocation of `JHOVE2.characterize()`.

         - In clumps, the child `Source` objects are created before the parser sees the aggregate. The dispatcher simply passes these child `Source` objects, along with the parent clump `Source`, to the aggregate format module, which makes the determination whether further parsing of these child Sources is required, or whether the information already extracted about the `Source` is sufficient.

      ii. If the module implements `Validator`, the `validate()` method is invoked to determine the source unit’s adherence to the general specifications for the formats.

   Note: The format module does not need to adhere to any strict distinction between parsing and validation. It may be more efficient to carry out some or all of the validation in the `parse()` method. In the extreme case, `validate()` simply reports the validation status.
iii. If any format profiles are defined for this format module, their validate() methods are invoked in order to perform more specialized validation. This step is managed by DispatcherCommand.

c. Digesting: If digest (checksum) calculation is specified on the command line or in the Spring configuration files, and the source unit represents a file or bytestream (all source units that aren’t file sets, directories, or clumps), each configured digest algorithm is invoked to create a message digest for the source unit. This step is managed by DigesterCommand.

d. Assessment: Assessment rules are applied to implement user- or institution-specific policy features of characterization. Assessment indicates the acceptability of a format instance, rather than, or in addition to, its conformance or non-conformance to a format specification. This step is managed by AssessorCommand.

e. Aggregation: If the source unit is a container or file set, the child source units are checked against patterns that identify clump source units. For each identified clump, the relevant child Source objects are “re-parented” to a new ClumpSource object, which replaces them as a child of the original parent source unit. The ClumpSource object is then passed to a recursive invocation of JHOVE2.characterize(). This step is managed by AggrefierCommand.

4. The Source object is passed to a displayer module, which filters, formats, and outputs the data obtained during the previous steps. The specific displayer module used is specified by the user on the command line.

Notice the recursive calls to JHOVE2.characterize() during the dispatch and aggregation steps. In this way, the application can process any number of source units, starting from a single (possibly aggregate) source unit.

As a source unit passes through this process, it accumulates a list of all the modules that have examined it. The modules in turn, record data in their reportable properties. In this way, the initial Source object becomes the root of a tree of Source objects, modules, and properties. The displayer’s job is to write this tree out in a useful form.

Characterization Example: an ASCII File
Let’s walk through the characterization of a simple non-remote ASCII file. If we run the jhove2 command line application with the file name licenses/LICENSE.txt as the sole argument, the following occurs

1. The jhove2 script invokes JHOVE2CommandLine.main() with the path to the ASCII file.
2. Seeing a single file path parameter, `main()` creates a `FileSource` object to encapsulate information related to `licenses/LICENSE.txt`. This information is accessible through the object’s reportable properties (see below, “Reportables and Modules”).

One particularly important property is `modules`, which records all modules used in the characterization process. Initially, the following modules are recorded:

- `JHOVE2CommandLine` itself.
- The `JHOVE2` core framework object itself (because `main()` is about to pass the `FileSource` object to `JHOVE2.characterize()`).
- The `Displayer` module that will be used after `characterize()` returns. This can be specified on the `jhove2` command line or in the Spring configuration file; it defaults to `TextDisplayer`, which we’ll assume for the rest of this example.

3. The `FileSource` object is passed to `JHOVE2.characterize()`. The following steps occur:

   a. Identification. This produces several presumptive format names in the PRONOM PUID namespace. These PUIDs are mapped to format names in the JHOVE2 namespace by the identification process:

      - `http://jhove2.org/terms/format/utf-8`
      - `http://jhove2.org/terms/format/utf-8/ascii`
      - `http://jhove2.org/terms/format/ibm-displaywrite`

      The following two modules are used in the identification process, and are added to the `FileSource` object’s `modules` property:

      - `IdentifierModule`.
      - `DROIDIdentifier`.

   b. Dispatch. The set of format modules that support the formats determined by the Identification step is determined. The `utf8` and `utf8/ascii` formats are supported by `UTF8Module` and no other. There is no format module for `ibm-displaywrite`, so `BaseFormatModule` is used to record this fact in a JHOVE2 `Message` field (JHOVE2 uses `Message` instances to record all information about difficulties encountered, whether created by difficulties in processing, or by anomalies in the format instance itself. The text of these messages is `internationalizable`, and can be found in file `config/messages/jhove2-messages.properties`).
In addition to recording the information about the unsupported format, the following steps are taken:

i. **UTF8Module.parse()** is invoked. This method parses the files in accordance with UTF-8 specifications and records file data. This includes a list of the Unicode code blocks used in the files. Parsing determines whether the file is valid UTF-8, and the module’s validity field is set accordingly.

ii. **UTF8Module** implements **Validator**, so its **validate()** method is invoked. Since actual validation was already done in the **parse()** method, the **validate()** method simply returns the value of the validity field.

iii. The **validate()** methods of any associated format profiles are invoked. Exactly one such profile exists for **UTF8Module: ASCIIProfile**. This module’s **validate()** method examines the list of Unicode code blocks created when the file was parsed. If the only code block listed is Basic Latin, then the file is considered valid ASCII.

The following modules are added to the **FileSource** object’s **modules** property:

- **UTF8Module**.
- **BaseFormatModule**.

**ASCIIProfile** is added to the **profiles** property of the **UTF8Module**.

c. Digest. Assuming the user has specified digest calculation on the command line or in the Spring configuration files, all configured digest algorithms are called, and the following module is added to the **FileSource** object’s **modules** property:

- **DigesterModule**.

d. Assessment. If any rules specify modules that match one of the modules already in the **FileSource** object’s **modules** property, these rules are evaluated. The following module is added:

- **AssessmentModule**.

e. Aggregation. Since the source unit has no children, there are no aggregation matches. If there had been one or more matches, **AggregfierModule** would have been added to the list.

4. The **FileSource** object is passed to the user-specified display module.
The data structure produced by this process is visualized in Figure 2. Each object is shown as a list of reportable properties. The root of the structure is the `FileSource` object, whose `modules` property refers to a list of modules. One of these modules, `UTF8Module`, has a `profiles` property that refers to a list containing a single profile, `ASCIIProfile`.

**Figure 2: Characterization Data for ASCII File**

---

**Characterization Example: Nested Containers**

This example is much more complex. Consider a file system directory containing four files: `Descriptions.htm`, `ts99_d00.dbf`, `ts99_d00.shp`, and `ts99_d00.shx`. The characterization process is very similar to that for the ASCII file example. The crucial difference here is that there are two places in the process where `JHOVE2.characterize()` is invoked recursively.
When this directory is characterized, the **DirectorySource** object is matched with the format module **DirectoryModule**, and **DirectoryModule.parse()** is invoked.

**Note:** **DirectoryModule** does not implement **Validator** and does not provide format profiles, so parsing is the only action taken during the dispatch step.

Parsing a directory consists simply of identifying the files within the directory, creating a **FileSource** object for each file, and passing each object to a recursive invocation of **JHOVE2.characterize()**. After the dispatch step is finished, the characterization data structure consists of a single **DirectorySource** object with four **FileSource** children (Figure 3).

This would be the final form of the data structure if the directory didn’t contain any clump source units. But it does: `ts99_d00.dbf`, `ts99_d00.shp`, and `ts99_d00.shx` constitute a GIS shapefile, which must be characterized as a unit.

The aggrefigation step accomplishes this. The aggrefier recognizes the clump source unit and performs these steps:

1. A new **ClumpSource** object is created, and made a child of the **DirectorySource** object.
2. The three **FileSource** objects are removed from the child list for the **DirectorySource** object and added to the child list for the **ClumpSource** object.
3. **JHOVE2.characterize()** is invoked recursively with the ClumpSource object as an argument.

This creates the final form of the characterization data structure (Figure 4).
Figure 3: Directory Characterization Example Data before Aggregation
Figure 4: Final Form of Directory Characterization Example
Reportables and Modules

Reportables are objects that use Reportable Properties to record and report characterization information. Reportable Properties are fields that come with a flexible data recording and discovery mechanism provided by JHOVE2 via the Java mechanisms of Annotation and Reflection. As you will see, you do not need to know the details about either of these mechanisms to employ them quite easily in your new module.

Modules are Reportables that encapsulate a particular process within the greater characterization process, with the results available as Reportable Properties.

Reportable classes are declared using the Spring Framework conventions for Spring Beans. Working with reportables does not require detailed knowledge of the Spring Framework. It does require knowledge of some Spring Framework conventions for defining and using beans and properties in Java code and in XML.

Reportables in Java Code

The Reportable interface is always implemented by descent from the AbstractReportable class, usually through one of its subclasses, such as AbstractModule.

Zero-Argument Constructor
For memory management reasons (see “Enforce Persistence Requirements”, below), reportable classes must always provide a zero-argument constructor.

Defining a Property
By convention, a property has a name and an accessor method (also known as a “getter”). If the property is not read-only, it also has a mutator method (“setter”). For example:

- processResult
- getProcessResult()
- setProcessResult()

Note the naming conventions: the property name is not capitalized, and uses medial capitals to indicate multiple words; the getter and setter use the capitalized property name preceded by get (non-Boolean getters), is (Boolean getters) or set (all setters). Property fields are defined with the protected access level modifier.
Figure 5 shows the getter and declaration for `processResult`. Note the `@ReportableProperty` annotation, which must appear immediately before the getter declaration. This annotation is what provides JHOVE2 with its flexible property discovery and display mechanism. It alerts the JHOVE2 Displayer classes that the contents of a field are to be reported in the JHOVE2 output.

**NOTE:** This annotation is also used by the JHOVE2 utility applications that create user-configurable properties files that control whether or not to display a reportable property, and what units of measure to associate with the reportable property. And it is used by the utility application that automatically generates information about a format module’s reportable properties for the specification document for the module. See the JHOVE2 wiki for more information about using these utilities, and about the use of Java annotations and reflection in JHOVE2.

Unless the property is read-only, a mutator method (setter) is also defined (Figure 5). The setter name must be `set` followed by the capitalized property name.

```
Figure 5: Property Definition with Annotation and Getter

@ReportableProperty(value = "Process Result")
public Result getProcessResult() {
    return this.processResult;
}

public void setProcessResult(Result processResult) {
    this.processResult = processResult;
}
```

The `@ReportableProperty` annotation has four possible parameters. Any parameter can be omitted, but you should always include the `value` parameter.

- **value**
  
  Always used. A *String* containing plain language description of the property.

- **order**
  
  Used whenever a class defines more than one property. Specifies the order in which properties are displayed. Order values do not need to be sequential. The placement of properties with missing or duplicate `order` parameters is undefined.

- **ref**
  
  Used by format modules for properties that comply with an official or informal specification. A *String* containing a reference to that specification. An example is in Figure 6.

```
Figure 6: Reportable Property Example with References

JHOVE2: Next-Generation Architecture for Format-Aware Characterization
```
@ReportableProperty(value="Profile tag table", ref="ICC.1:2004-10, §7.7.3")
public ICCTagTable getTagTable()
{
    return this.tagTable;
}

type

Used in format modules that define a pair of properties for a single characteristic of a source unit. One property has type=PropertyType.Raw, indicating a literal property value from the source unit. The other property has type=PropertyType.Descriptive, indicating that the value has been interpreted by the format module.

When a single property is provided, type is omitted. Display modules can be configured so that only raw properties are output, only descriptive properties are output, or both. This configuration does not affect properties with the default type.

Inherited Properties

Reportables inherit any properties defined in their superclasses and interfaces. These inherited properties are output after the class’s own properties. The order parameter designates the order of output relative to other properties from the same interface.

All Reportables inherit the reportableName property, defined in AbstractReportable. This string property is automatically set to the unqualified name of the Reportable’s class, and cannot be changed. This property is also output last.

Spring Configuration of Reportables

Note: This section describes conventions that are followed in most of the JHOVE2 configuration files. It is designed for programmers who are not Spring Framework experts. For a full description of the allowed XML used in bean definitions, refer to the Spring Framework Reference Documentation, “The IoC container” (http://static.springsource.org/spring/docs/3.0.x/spring-framework-reference/html/beans.html).

Note: Use of a validating XML editor is strongly recommended. Eclipse, NetBeans, and several other IDEs include such an editor.

One of the key architectural idioms of JHOVE2 is Inversion of Control/Dependency Injection. At large scale, this means components like Commands and Modules can be plugged into the JHOVE2 framework to customize the sequence of commands invoked, to add new format modules, or, for example, to replace the out-of-box DROID identification module with a different one. At finer scale, JHOVE2 uses
the Spring Framework’s dependency injection features to create instances of Formats, Modules, Profiles, Commands, and other Reportables.

Reportable classes can be instantiated in the usual way, by invoking the class’s constructor. The JHOVE2 distribution configures instances in XML files, from which the Spring framework instantiates objects as they are needed.

Each XML file must appear in the directory hierarchy under config, and must have a name that fits the pattern jhove2-id-config.xml, where id is any string valid in a file name.

These are the only formal name and location constraints, but for the sake of maintainability, files are organized into a directory structure under config/spring that mirrors the package structure of the Java source code. Thus the UTF8 Format Module, which is defined in org.jhove2.module.format.utf8, has its XML files in config/spring/module/format/utf8. Each file contains a single beans element, which must be defined as shown in Figure 7.

**Figure 7: beans Stub for XML Configuration file**

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:util="http://www.springframework.org/schema/util"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
    http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-context-2.5.xsd
    http://www.springframework.org/schema/util
    http://www.springframework.org/schema/util/spring-util-2.0.xsd">
    <!--individual bean definitions go here -->
</beans>
```

In Figure 8, we have a simple bean definition for an I8R object. This matches the constructor that takes a single String parameter, indicated by the nested constructor-arg element. Note the scope attribute, which indicates that the bean is a singleton. This means that the object is only instantiated the first time a bean with ID UTF8Identifier is need; each subsequent reference is to the existing object.

Many beans define objects such as format modules designed to accumulate characterization data. These have scope="prototype", indicating that a new object is instantiated every time the bean is referenced.

The XML tells the framework how to construct an I8R object and how to define its value and namespace properties. These properties are read-only, so they must be specified by the constructor.
When the property is read-write, it can be set by the nested **property** element (Figure 9). Some constructor arguments properties are collections; these are specified with nested **set** or **list** elements (Figure 10). Constructor arguments and property values can be references to other beans (Figure 11).

**Figure 8: Simple Bean Defined by Constructor Argument**

```xml
<bean id="UTF8Identifier" class="org.jhove2.core.I8R" scope="singleton">
    <constructor-arg type="java.lang.String"
        value="http://jhove2.org/terms/format/utf-8"/>
</bean>
```

**Figure 9: Bean with Properties**

```xml
<bean id="WAVESpecification" class="org.jhove2.core.Document"
    scope="singleton">
    <constructor-arg type="java.lang.String"
        value="Multimedia Programming Interface and Data Specifications"/>
    <constructor-arg type="org.jhove2.core.Document$Type"
        value="Specification"/>
    <constructor-arg type="org.jhove2.core.Document$Intention"
        value="Authoritative"/>
    <property name="date" value="August 1991"/>
    <property name="edition" value="Version 1.0"/>
    <property name="publisher"
        value="IBM Corporation/Microsoft Corporation"/>
</bean>
```

**Figure 10: List Property**

```xml
<property name="articles">
    <list value-type="java.lang.String">
        <value>a</value>
        <value>an</value>
        <value>the</value>
    </list>
</property>
```

**Figure 11: Bean References**

```xml
<constructor-arg ref="BytestreamIdentifier"/>
<property name="configInfo" ref="ConfigInfo"/>
<property name="specifications">
    <list value-type="org.jhove2.core.Document">
        <ref bean="UnicodeSpecification"/>
        <ref bean="ISO10646Standard"/>
        <ref bean="RFC3629RFC"/>
    </list>
</property>
```

JHOVE2: Next-Generation Architecture for Format-Aware Characterization
Note the use of data types in the preceding examples. They may seem inconsistent, but all follow rules that are designed to make life easy for the programmer who is not a Spring expert. The actual rules imposed by Spring 3.0 are more flexible.

The rules used here are as follows

- All type names are fully-qualified (`java.lang.String`, not `String`).

- Constructor arguments:
  - When the `value` attribute is used (indicating a non-reference argument), always provide a `type` attribute specifying the type.
  - When the `ref` attribute is used (indicating a reference argument), do not specify the type.

- Properties:
  - When a non-collection value is specified by a `value` or `ref` attribute, do not specify a type.
  - When a collection value is specified by a nested `set` or `list` element, the nested element always provides a `value-type` parameter specifying the type of the collection contents.

**NOTE:** You can see in more detail how JHOVE2 uses Spring configuration by exploring the various factory classes in package `org.jhove2.config.spring`. 
Format Modules, Format Profiles, and Formats

This section describes the creation of format modules, format profiles, and formats. These are the objects that perform format-specific characterization.

Format modules perform the initial characterization of all source units; they are often designed around a “family” of related formats. If more specific characterization is called for, a format profile, specializing in a specific format within the family, is used.

The UTF8Module format module and ASCIIProfile format profile are useful examples of this division of labor. Excerpts from the Java and XML code for these two modules are used throughout this section, sometimes in simplified form.

This section is organized as follows:

- “Creating a Format Module Class” on page 22.
- “Creating a Format Profile Class” on page 27.

Creating a Format Module Class

Creating a format module class consists of the following tasks:

- “Define the Class” on page 23.
- “Define Fields” on page 23.
- “Implement Properties” on page 24.
- “Implement Constructors” on page 24.
- “Implement a Parser” on page 24.
- “Implement Validator” on page 25.

Additional steps are required for parsing aggregates, as described in “Parsing Aggregate Source Units” on page 26.
Define the Class

All format module classes extend `BaseFormatModule`, which implements the `FormatModule` interface. If the class does any validation, it must also implement `Validator` (Figure 12).

```
Figure 12: UTF8 Module Class Declaration

public class UTF8Module
    extends BaseFormatModule
    implements Validator
```

The package for the class should go into a sub-package of `org.jhove2.module.format`. The `UTF8Module` class can be found in package `org.jhove2.module.format.utf8`, along with any profile or other supporting classes for that module.

Define Fields

Format modules should begin with declaration of the `VERSION`, `RELEASE`, and `RIGHTS` constants (Figure 13). These will be needed later in the constructor code. As with all Reportables, Format Modules need to store the values of their properties. This is typically done in instance field variables (Figure 14).

Format modules that implement `Validator` have a `coverage` property, indicating whether validation is inclusive (covers all features of the format) or selective (covers only some features). This value represents an implementation detail, so it is stored as a constant (Figure 15).

```
/** UTF-8 module version identifier. */
public static final String VERSION = "2.0.0";

/** UTF-8 module release date. */
public static final String RELEASE = "2010-09-10";

/** UTF-8 module rights statement. */
public static final String RIGHTS =
    "Copyright 2010 by The Regents of the University of California. " +
    "Available under the terms of the BSD license."

/** Number of characters. */
protected long numCharacters;

/** UTF-8 module validation coverage. */
public static final Coverage COVERAGE = Coverage.Inclusive;
```
Implement Properties
Format modules usually have properties that report characterization information. To implement properties, refer to “Reportables in Java Code” on page 16.

Implement Constructors
The constructor must initialize the superclass. Typically, the constructor will also initialize variable fields, including those that hold property values.

As with all reportables, format modules must have a zero-argument constructor. Additionally, each format module must have a two-argument constructor. The first argument is the Format object with information, including format aliases, for the format or format family which this module is constructed to parse. The second argument is a FormatModuleAccessor object, a utility object that manages persistence for format modules (see “Enforce Persistence Requirements”, below).

Figure 16: UTF8Module Constructors

```java
public UTF8Module(Format format, FormatModuleAccessor formatModuleAccessor) {
    super(VERSION, RELEASE, RIGHTS, format, formatModuleAccessor);
    this.c0Characters      = new TreeSet<C0Control>();
    this.c1Characters      = new TreeSet<C1Control>();
    this.codeBlocks        = new TreeSet<CodeBlock>();
    this.eolMarkers        = new TreeSet<EOL>();
    this.invalidCharacters = new ArrayList<UTF8Character>();
    this.isValid           = Validity.Undetermined;
    this.numCharacters     = 0L;
    this.numLines          = 0L;
    this.numNonCharacters  = 0L;
}
public UTF8Module(){
    this(null, null);
}
```

Implement a Parser
Format modules always implement the Parser interface. The parse() method is invoked with three arguments:

- The global JHOVE2 object that represents the framework as a whole.
- A Source object that represents the source unit.
- An Input object that encapsulates an input buffer and provides information about the file being read.
The beginning of `parse()` in `UTF8Module` is shown in Figure 17. Notice that the `start` and `end` variables are set using the `Source` object properties, not the `Input` object properties. This is necessary because the underlying file and the source unit are not necessarily the same thing; the source unit could be a bytestream in a larger file.

_Figure 17: Beginning of UTF8Module.parse()

```java
public long parse(JHOVE2 jhove2, Source source, Input input)
    throws EOFException, IOException, JHOVE2Exception
{
    long consumed = 0L;
    this.isValid = Validity.Undetermined;
    int numErrors = 0;

    long start =         ((MeasurableSource) source).getStartingOffset();
    long end   = start + ((MeasurableSource) source).getSize();
    input.setPosition(start);
```

Implement Validator

Validation by format modules is not mandatory. Implementing the `Validator` interface tells the JHOVE2 framework that the format module performs validation, so that `validate()` is invoked and validation results are output.

The `Validator` interface consists of two properties (and their getter methods) and a single non-property method:

- coverage
  Format coverage for validation. Can be `Coverage.inclusive` (validation covers all features of the format) or `Coverage.selective` (validation covers only selected features). Since `coverage` describes the implementation of the module, it normally reports the value of a constant.

- valid
  Validation result. Can be `Validity.True`, `Validity.False`, or `Validity.Undetermined`.

- validate()
  Invoked automatically by the framework. This method takes the same arguments as `parse()` and returns the result of validation. Note that `validate()` does not necessarily perform the validation itself; in many cases (such as `UTF8Module`) it makes sense to perform validation concurrently with parsing. In these cases, validation is done in `parse()`, and `validate()` simply returns the value of the `valid` property.
Enforce Persistence Requirements

When processing a single file source, or a directory or container file such as a ZIP file that does not contain very many children (other files, directories, etc.), JHOVE2 can keep all the information it accumulates about those sources in the application’s short term memory. However, if a large number of sources or a deeply nested container such as a directory or ZIP file is being processed, JHOVE2 has to augment its short-term memory by "persisting" the information to disk until it has completed processing all the sources, and written all the output about them. JHOVE2 can be configured to use either an in-memory or persistent memory manager.

JHOVE2 uses BerkeleyDB JE as its persistence memory manager. Whether JHOVE2 is configured to use short-term memory or persistent memory, the Reportable classes you create as part of your format module have to conform to the requirements of BerkeleyDB JE Persistent classes.

This means, in addition to the FormatModuleAccessor mentioned above in the two-argument FormatModule constructor, that you must follow certain other constraints of BerkeleyDB JE-persistable classes:

- The module class declaration must be annotated with the BerkeleyDB JE @Persistent annotation

    ```java
    @Persistent
    public class UTF8Module
    {
        extends BaseFormatModule
        implements Validator
    }
    ```

- The module must have a 0-argument constructor

- The module should not contain any non-static nested (inner) classes

- All module field types must

  - be a "simple" Java type or
  - be another @Persistent type or
  - have a `com.sleepycat.persist.model.PersistentProxy` implementation created for it in package `org.jhove2.persist.berkeleydpl.proxies`

You can find more information about the use of BerkeleyDB JE in JHOVE2 on the Memory Management page of the JHOVE2 wiki.

Parsing Aggregate Source Units

When parsing an aggregate source unit, a format module must initiate the characterization of each child source unit. This is done by invoking JHOVE2.characterize(), following this pattern:
jhove2.characterize(source, input)

where jhove2 and input are the JHOVE2 and Input objects that were passed to parse(), and source is the Source object representing the child.

If the aggregate is a file set or a directory, then the Source object passed to parse() has a childSources property that already refers to a List of child Source objects. In this case, the format module’s job is to retrieve these child objects by invoking Source.getChildSources() and then pass them one by one to JHOVE2.characterize().

However, if the aggregate is a container (such as a ZIP archive) or a wrapper (such as a TIFF file) then the format module must set the childSources property itself. That means it must instantiate a Source object for each child source unit it discovers as it parses the source unit.

The JHOVE2 framework object comes with a SourceFactory utility member, with factory methods for instantiating Source objects. You should use one of these factory methods, instead of the protected Source constructors, to create your Source objects.

If the aggregate source is a clump source (detected and created, with its child Sources attached, by the AggregferModule), the aggregate format module to which the clump Source is dispatched via the recursive invocation of the JHOVE2.characterize() method will determine if any further parsing of child Source objects is required, or if all that is needed is inspection of Reportable properties whose values were set by the parse of the individual child Sources.

Creating a Format Profile Class

After a format module is finished with a source unit, any format profiles associated with the format module are invoked to evaluate conformance to format profiles. While format profile classes are very similar to format profile classes, there are the important differences:

- Format module classes extend AbstractFormatProfile, which implements FormatProfile.
- There is no parse() method.
- Validation is always performed. FormatProfile is an extension of Validator, so there must be a validate() method along coverage and valid properties.

ASCIIProfile is a simple example of a format profile. It is a profile for the UTF-8 format; its job is to examine the code blocks discovered by UTF8Module and report the ones that are not in the Basic Latin code block. If it discovers any such blocks, it sets the valid property to Validity.False. Otherwise, valid is set to Validity.true.
As with FormatModule classes, FormatProfile classes must be annotated with the @Persistent annotation, and conform to the other requirements for BerkeleyDB JE persistent classes (see “Enforce Persistence Requirements” above). The two-argument constructor for a FormatProfile object includes an argument for a FormatProfileAccessor object to manage FormatProfile object persistence.

```java
public ASCIIProfile(Format format, FormatProfileAccessor formatProfileAccessor) {
    super(VERSION, RELEASE, RIGHTS, format, formatProfileAccessor);
    this.isValid = Validity.Undetermined;
    this.nonBasicLatinCodeBlocks = new TreeSet<CodeBlock>();
}
```

**Using Spring to Instantiate Format Modules, Format Profiles, and Formats**

The JHOVE2 framework relies on Spring beans, defined in XML, to specify the way in which format modules, format profiles, and their ancillary objects are created. Coding the XML consists of these tasks:

- “Define the Beans for the Format Module and Format Profile” on page 28.
- “Define the Beans for Format Objects” on page 30.

**Define the Beans for the Format Module and Format Profile**

The format module bean instantiates the format module class. FormatModule constructors require a Format object argument and a FormatModuleAccessor object, so we specify a constructor argument that references the appropriate bean definitions. Since there needs to be a unique instance for each characterized source unit, we specify a prototype bean.

Figure 18 shows a minimal bean definition for an instance of UTF8Module, constructed with a reference to a Format object and a FormatModuleAccessor object.

Format module beans commonly set some of the properties of the object. In Figure 19, we see a more complete definition of a UTF8Module instance, with two property values:

- The profiles property, defined in the Format interface, contains a list of format profiles for this format module. Here, our instance of UTF8Module has a single format profile, an instance of ASCIIProfile.

- The developers property, defined in the Module interface, uses an Agent instance to document the authors of the bean class. This is information that gets reused in a number of bean definitions, so it is centrally located in a file in the config directory, rather than the class-specific directory. As with all “constant” objects, it is defined as a singleton.
Note: The Module interface defines a number of properties similar to developers, designed to help document module classes.

Beans for format profiles are similar to those for format modules. In Figure 21 we see the bean for the instance of ASCIIProfile referenced in Figure 19.

Figure 18: Minimal Bean for UTF8Module

```xml
<bean id="UTF8Module" class="org.jhove2.module.format.utf8.UTF8Module"
     scope="prototype">
    <constructor-arg ref="UTF8Format"/>
    <constructor-arg ref="FormatModuleAccessor"/>
</bean>
```

Figure 19: More Complete Bean for UTF8Module

```xml
<bean id="UTF8Module" class="org.jhove2.module.format.utf8.UTF8Module"
     scope="prototype">
    <constructor-arg ref="UTF8Format"/>
    <constructor-arg ref="FormatModuleAccessor"/>
    <property name="profiles">
        <list value-type="org.jhove2.module.format.FormatProfile">
            <ref bean="ASCIIProfile"/>
        </list>
    </property>
    <property name="developers">
        <list value-type="org.jhove2.core.Agent">
            <ref bean="CDLAgent"/>
        </list>
    </property>
</bean>
```

Figure 20: Agent Bean

```xml
<bean id="CDLAgent" class="org.jhove2.core.Agent" scope="singleton">
    <constructor-arg type="java.lang.String"
                     value="California Digital Library"/>
    <constructor-arg type="org.jhove2.core.Agent$Type" value="Corporate"/>
    <property name="URI" value="http://www.cdlib.org/"/>
</bean>
```

Figure 21: Bean for ASCIIProfile

```xml
<bean id="ASCIIProfile"
     class="org.jhove2.module.format.utf8.ascii.ASCIIProfile" scope="prototype">
    <constructor-arg ref="ASCIIFormat"/>
    <constructor-arg ref="FormatProfileAccessor"/>
</bean>
```
Define the Beans for Format Objects

Whereas format modules and format profiles are mutable objects that accumulate characterization data relating to specific formats, format objects are constant objects that specify information about the formats themselves.

A big part of the format object’s job is recording a format’s various identifiers. Each identifier is represented by an I8R object. (The class name is chosen to distinguish format identifiers from Identifier modules; see “The Characterization Process” on page 6.) The I8R instance encapsulates a String identifier together with a Namespace value associated with the identifier. In Table 1, we see some I8R values associated with UTF-8 files.

The Format constructor takes an I8R object argument which represents the JHOVE2 identifier for the format. Additional format identifiers are specified in the aliasIdentifiers property. Figure 22 shows a minimal format object bean that might be used with the minimal format module bean in Figure 18. This format bean associates the JHOVE2 and PRONOM identifiers for UTF-8 as two names for the same thing. By specifying this Format object as the argument for the UTF8Module constructor, the XML informs the framework that UTF8Module is the format module to use when a source unit with PRONOM PUID “x-fmt/16” is identified.

The constructor for Format takes the following arguments:

- A String giving the JHOVE2 name for the format.
- An I8R object for the JHOVE2 identifier.
- A Type value that gives the scope of the format. Type.Family is used with basic format objects that are passed to format module constructors. Type.Format is used with more specialized format objects that are passed to format profile constructors.
- An Ambiguity value that can be Ambiguity.Ambiguous or Ambiguity.Unambiguous.

A more realistic example of a UTF-8 format object bean is shown in Figure 23, which might be used with the format module bean in Figure 19. This version has the following additions:

- A more complete list if identifiers for the UTF-8 format.
- The aliasNames property is used to document alternative names for the format.
- The **specifications** property is used to document the specifications the format object covers.

The ASCII format profile bean in Figure 21 also needs a format object bean (Figure 24). Notice the similarity to the UTF-8 format object bean, but with the third constructor argument changed to `Format`.

### Table 1: Selected I8R Values for UTF-8 Files

<table>
<thead>
<tr>
<th>Value</th>
<th>Namespace</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><a href="http://jhove2.org/terms/format/utf-8">http://jhove2.org/terms/format/utf-8</a></td>
<td>Namespace.JHOVE2</td>
<td>JHOVE2 identifier</td>
</tr>
<tr>
<td>x-fmt/16</td>
<td>Namespace.PUID</td>
<td>PRONOM Persistent Unique Identifier</td>
</tr>
<tr>
<td>text/plain; charset=&quot;UTF-8&quot;</td>
<td>Namespace.MIME</td>
<td>MIME type</td>
</tr>
<tr>
<td>public.utf8-plain-text</td>
<td>Namespace.UTI</td>
<td>Apple Uniform Type Identifier</td>
</tr>
<tr>
<td>RFC 3629</td>
<td>Namespace.RFC</td>
<td>IETF Request For Comments</td>
</tr>
</tbody>
</table>

**Figure 22: Minimal UTF-8 Format Bean with I8R Beans**

```xml
<bean id="UTF8Format" class="org.jhove2.core.format.Format"
      scope="singleton">
  <constructor-arg type="java.lang.String" value="UTF-8"/>
  <constructor-arg ref="UTF8Identifier"/>
  <constructor-arg type="org.jhove2.core.format.Format$Type"
                  value="Family"/>
  <constructor-arg type="org.jhove2.core.format.Format$Ambiguity"
                  value="Unambiguous"/>
  <property name="aliasIdentifiers">
    <set value-type="org.jhove2.core.I8R">
      <ref bean="PUIDxfmt16"/>
    </set>
  </property>
</bean>

<bean id="UTF8Identifier" class="org.jhove2.core.I8R" scope="singleton">
  <constructor-arg type="java.lang.String"
                 value="http://jhove2.org/terms/format/utf-8"/>
</bean>

<bean id="PUIDxfmt16" class="org.jhove2.core.I8R" scope="singleton">
  <constructor-arg type="java.lang.String" value="x-fmt/16"/>
  <constructor-arg type="org.jhove2.core.I8R$Namespace" value="PUID"/>
</bean>
```
Figure 23: Complete Format Bean for UTF-8

```xml
<bean id="UTF8Format" class="org.jhove2.core.format.Format" scope="singleton">
  <constructor-arg type="java.lang.String" value="UTF-8"/>
  <constructor-arg ref="UTF8Identifier"/>
  <constructor-arg type="org.jhove2.core.format.Format$Type" value="Family"/>
  <constructor-arg type="org.jhove2.core.format.Format$Ambiguity" value="Unambiguous"/>
  <property name="aliasIdentifiers">
    <set value-type="org.jhove2.core.I8R">
      <ref bean="UTF8Charset"/>
      <ref bean="UTF8MIMEType"/>
      <ref bean="UTF8RFC3629"/>
      <ref bean="UTF8UTI"/>
      <ref bean="PUIDxfmt13"/>
      <ref bean="PUIDxfmt16"/>
      <ref bean="PUIDxfmt110"/>
      <ref bean="PUIDxfmt111"/>
    </set>
  </property>
  <property name="aliasNames">
    <set>
      <value>UCS Transformation Format, 8 bit</value>
      <value>Unicode Transformation Format, 8 bit</value>
    </set>
  </property>
  <property name="specifications">
    <list value-type="org.jhove2.core.Document">
      <ref bean="UnicodeSpecification"/>
      <ref bean="ISO10646Standard"/>
      <ref bean="RFC3629RFC"/>
    </list>
  </property>
</bean>
```
Figure 24: Bean for ASCII Format

```xml
<bean id="ASCIIFormat" class="org.jhove2.core.format.Format" scope="singleton">
  <constructor-arg type="java.lang.String" value="ASCII"/>
  <constructor-arg ref="ASCIIIdentifier"/>
  <constructor-arg type="org.jhove2.core.format.Format$Type" value="Format"/>
  <constructor-arg type="org.jhove2.core.format.Format$Ambiguity" value="Unambiguous"/>
  <property name="aliasIdentifiers">
    <set value-type="org.jhove2.core.I8R">
      <ref bean="ASCIIANSIX34"/>
      <ref bean="ASCIICharset"/>
      <ref bean="ASCIICharsetUS"/>
      <ref bean="ASCIIECMA6"/>
      <ref bean="ASCIIISO646"/>
      <ref bean="ASCIITUT50"/>
      <ref bean="ASCIIMIMEType"/>
      <ref bean="ASCIIRFC1345"/>
      <ref bean="PUIDxfmt14"/>
      <ref bean="PUIDxfmt15"/>
      <ref bean="PUIDxfmt21"/>
      <ref bean="PUIDxfmt22"/>
      <ref bean="PUIDxfmt130"/>
    </set>
  </property>
</bean>
```

NOTE: For each format module that is part of the JHOVE2 distribution, a complete set of Spring Beans have been created for the Format associated with that module, in the Spring configuration file for the module.

In addition, a minimal Spring Format bean has been created for each Format that has a DROID (PRONOM) PUID in the DROID signature file that is part of the JHOVE2 distribution, for which the distribution does NOT contain a format module that “understands” that format. These can be found in the `config/spring/module/format` directory, in file `jhove2-otherFormats-config.xml`. As you create new format modules for any of these formats, you can use and expand these beans with any additional information you have (about aliases, standards, etc.).
Where to Find More Information

The JHOVE2 Users Guide can be found on the JHOVE2 wiki.

The specifications for individual format modules can be found on the JHOVE2 wiki on the JHOVE2 Modules page.

Sequence and class diagrams can be found on the JHOVE2 wiki on the UML Diagrams page.

A step-by-step presentation on “Creating a New JHOVE2 Format Module,” including directions on creating JUnit tests, displayer and units of measure properties files, and format module specification documentation, can be found on the JHOVE2 wiki on the Public Presentations page.

Templates for creating Spring configuration files for a new JHOVE2 module, and for creating the documentation for the module, can be found on the JHOVE2 wiki on the Developers’ Templates page.
A Format Module Example

```java
package org.jhove2.module.format.xyzzy;

import java.io.EOFException;
import java.io.IOException;
import org.jhove2.annotation.ReportableProperty;
import org.jhove2.core.JHOVE2;
import org.jhove2.core.JHOVE2Exception;
import org.jhove2.core.format.Format;
import org.jhove2.core.io.Input;
import org.jhove2.core.source.Source;
import org.jhove2.module.format.BaseFormatModule;
import org.jhove2.module.format.Validator;
import com.sleepycat.persist.model.Persistent;

@Persistent
public class XyzzyModule
    extends BaseFormatModule
    implements Validator
{
    public static final String VERSION = "2.0.0";
    public static final String RELEASE = "2010-11-10";
    public static final String RIGHTS = "Copyright (c) Regents of the University of California";
    public static final Coverage COVERAGE = Coverage.Inclusive;

    /** Source unit validity. */
    private Validity validity;

    /**
     * Zero-argument constructor.
     */
    private XyzzyModule()
    {
        this(null, null);
    }

    /**
     * Xyzzy module constructor.
     */
    @param format Xyzzy format
    @param formatModuleAccessor
```
public XyzzyModule(Format format, FormatModuleAccessor formatModuleAccessor) {
    super(VERSION, RELEASE, RIGHTS, format);
    this.validity = Validity.Undetermined;
}

/**
* @see Parser#parse(org.jhove2.core.JHOVE2,
org.jhove2.core.source.Source, org.jhove2.core.io.
Input)
*/
@Override
public long parse(JHOVE2 jhove2, Source source, Input input) throws EOFException, IOException, JHOVE2Exception {
    long consumed = 0L;
    return consumed;
}

/**
* @see Validator#validate(org.jhove2.core.JHOVE2,
org.jhove2.core.source.Source, org.jhove2.core.io.
Input)
*/
@Override
public Validity validate(JHOVE2 jhove2, Source source, Input input) throws JHOVE2Exception {
    return this.validity;
}

/**
* Read-only Xyzzy Property
*/
@ReportableProperty(value = "Xyzzy property.")
public String getXyzzy() {
    return "yzzyx";
}

/**
* @see Validator#getCoverage()
*/
@Override
public Coverage getCoverage() {
    return COVERAGE;
}
```java
/**
 * @see Validator#isValid()
 */
@Override
public Validity isValid()
{
    return this.validity;
}
```
<set value-type="org.jhove2.core.I8R">  
  <ref bean="PUIDxfmt32767"/>  
</set>  
</property>  
</bean>

<!-- Xyzzy identifier bean -->  
<bean id="XyzzyIdentifier" class="org.jhove2.core.I8R" scope="singleton">  
  <constructor-arg type="java.lang.String" value="http://jhove2.org/terms/format/xyzzy"/>  
</bean>

<!-- PUID x-fmt/32767 aliasIdentifier bean (not a real PUID) -->  
<bean id="PUIDxfmt32767" class="org.jhove2.core.I8R" scope="singleton">  
  <constructor-arg type="java.lang.String" value="x-fmt/32767"/>  
  <constructor-arg type="org.jhove2.core.I8R$Namespace" value="PUID"/>  
</bean>
</beans>
Package Index

AbstractFormatProfile    org.jhove2.format
AbstractModule          org.jhove2.module
AbstractReportable      org.jhove2.core.reportable
Agent                   org.jhove2.core
AggregrefierCommand     org.jhove2.module.aggrefy
AggregrefierModule      org.jhove2.module.aggrefy
Ambiguity               org.jhove2.core.format.Format
ASCIIProfile            org.jhove2.module.format.utf8.ascii
AssessmentModule        org.jhove2.module.assess
AssessorCommand         org.jhove2.module.assess
BaseFormatModule        org.jhove2.module.format
ClumpSource             org.jhove2.core.source
Coverage                org.jhove2.module.format.Validator
DigesterCommand         org.jhove2.module.digest
DigesterModule          org.jhove2.module.digest
DirectoryModule         org.jhove2.module.format.directory
DirectorySource         org.jhove2.core.source
DispatcherCommand       org.jhove2.module.format
Displayer               org.jhove2.module.displayer
DROIDIdentifier         org.jhove2.module.identify
FileSetSource           org.jhove2.core.source
FileSource              org.jhove2.core.source
Format                  org.jhove2.core.format
FormatModule            org.jhove2.module.format
FormatProfile           org.jhove2.format
I8R                     org.jhove2.core
Identifier              org.jhove2.module.identify
IdentifierCommand       org.jhove2.module.identify
IdentifierModule        org.jhove2.module.identify
Input                   org.jhove2.core.io
JHOVE2                  org.jhove2.core
JHOVE2CommandLine       org.jhove2.app
List                    java.util
Module                  org.jhove2.module
Namespace               org.jhove2.core.I8R
PropertyType            org.jhove2.annotation.ReportableProperty
Reportable              org.jhove2.core.reportable
ReportableProperty      org.jhove2.annotation
Source                  org.jhove2.core.source
SourceFactory           org.jhove2.core.source
String                  java.lang
TextDisplayer           org.jhove2.module.displayer
Type                    org.jhove2.core.format.Format
<table>
<thead>
<tr>
<th>Class</th>
<th>Package</th>
</tr>
</thead>
<tbody>
<tr>
<td>URLSource</td>
<td>org.jhove2.core.source</td>
</tr>
<tr>
<td>UTF8Module</td>
<td>org.jhove2.module.format.utf8</td>
</tr>
<tr>
<td>Validator</td>
<td>org.jhove2.format</td>
</tr>
<tr>
<td>Validity</td>
<td>org.jhove2.format.Serializer</td>
</tr>
</tbody>
</table>