JavaBeans™ Activation Framework Specification *Version 1.0*

Bart Calder, Bill Shannon

This is a draft of the JavaBeans™ Activation Framework Specification, a proposed data typing and registry technology that will be released as a Standard Extension to the Java™ Platform.

1.0 Overview

JavaBeans™ is proving to be a popular technology. As more people embrace JavaBeans™ and the Java™ platform, some of the environment's shortcomings are brought to light. JavaBeans™ were meant to satisfy needs in builder and development environments but its capabilities fall short of those needed to deploy stand alone components as content editing and creating entities.

Neither JavaBeans™ nor the Java™ platform define a consistent strategy for typing data, a method for determining the supported data types of a software component, a method for binding typed data to a component, or an architecture and implementation that supports these features.

Presumably with these pieces in place, a developer can write a JavaBeans™ based component that provides helper application like functionality in a web browser, added functionality to an office suite, or a content viewer in a Java Station™ environment.

2.0 Goals

This document describes the JavaBeans™ Activation Framework (JAF). The JAF implements the following services:

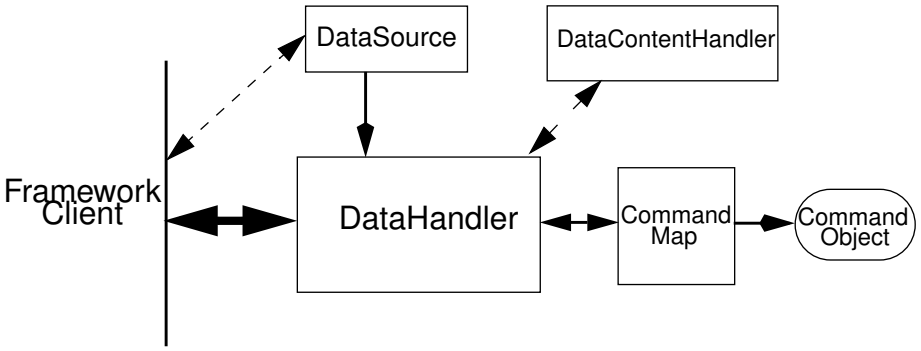
- It determines the type of arbitrary data.
- It encapsulates access to data.
- It discovers the operations available on a particular type of data.

- It instantiates the software component that corresponds to the desired operation on a particular piece of data.

The JAF is packaged as a Standard Extension to the Java™ platform.

3.0 Architectural Overview

JDK™ 1.1 (including JavaBeans™) already provides some support for a modest activation framework. The JAF leverages as much of that existing technology as possible. The JAF integrates these mechanisms.



This diagram shows the major elements comprising the JAF architecture. Note that the framework shown here is not bound to a particular application.

3.1 The DataHandler Class

The DataHandler class (shown in the diagram above) provides a consistent interface between JAF-aware clients and other subsystems.

3.2 The DataSource Interface

DataSource interface encapsulates an object that contains data, and that can return both a stream providing data access, and a string defining the MIME type describing the data.

Classes can be implemented for common data sources (web, file system, IMAP, ftp etc.). The DataSource interface can also be extended to allow per data source user customizations. Once the DataSource is set in the DataHandler, the client can determine the operations available on that data.

The JAF includes two DataSource class implementations for convenience:

- FileDataSource accesses data held in a file.
- URLDataSource accesses data held at a URL.

3.3 The CommandMap Interface

The CommandMap provides a service that allows consumers of its interfaces to determine the 'commands' available on a particular MIME Type as well as an interface to retrieve an object that can operate on an object of a particular MIME Type (effectively a component registry). The Command Map can generate and maintain a list of available capabilities on a particular data type by a mechanism defined by the implementation of the particular instance of the CommandMap.

The JavaBeans™ package provides the programming model for the software components that implemented the commands. Each JavaBean™ can use externalization, or can implement the *CommandObject* interface to allow the typed data to be passed to it.

The JAF defines the CommandMap interface, which provides a flexible and extensible framework for the CommandMap. The CommandMap interface allows developers to develop their own solutions for discovering which commands are available on the system. A possible implementation can access the 'types registry' on the platform or use a server-based solution. The JAF provides a simple default solution based on RFC 1524 (.mailcap) like functionality. See "Deliverables" below.

3.4 The Command Object Interface

JavaBeans™ extend the CommandObject interface in order to interact with JAF services. JAF-aware JavaBean™ can directly access their DataSource and DataHandler objects in order to retrieve the data type, and to act on the data.

4.0 Using The Framework

We intend to make this infrastructure widely available for any Java™ Application that needs this functionality. The 'canonical' consumer of this framework accesses it through the DataHandler (although the major subsystems are designed to also operate independently). An underlying DataSource object is associated with the DataHandler when the DataHandler class is constructed.

- The DataHandler retrieves the data typing information from the DataSource or gets the data type directly from the constructor.
- Once this initialization step is complete, request a list of commands that can be performed on the data item can be accessed from the DataHandler.

When an application issues a request for this list, the DataHandler uses the MIME data type specifier returned, in order to request a list of available commands from the CommandMap object. The CommandMap has knowledge of available commands (implemented as JavaBeans™) and their supported data types. The CommandMap returns a subset of the full list of all commands based on the requested MIME type and the semantics of the CommandMap implementation, to the DataHandler.

Ultimately when the application wishes to apply a command to some data, it is accomplished through the appropriate DataHandler interface which uses the CommandMap to retrieve the appropriate JavaBean™ which is used to operate on the data. The container (user of the framework) makes the association between the data and the JavaBean™.

5.0 Usage Scenarios

This scenario uses the example of a hypothetical file viewer application in order to illustrate the normal flow of tasks involved when implementing the JAF. The file viewer is similar to CDE's 'dtfile,' or to the Windows 95 Explorer utility. When launched, it presents the user with a display of available files. It includes a function like CDE's dtfile or Explorer's 'right mouse' menu, where all operations that can be performed on a selected data item are listed in a popup menu for that item.

A typical user launches this application to view a directory of files. When the user specifies a file by clicking on it, the application displays a popup menu which lists the available operations on that file. File system viewer utilities normally include 'edit,' 'view,' and 'print' commands as available operations. For instance selecting 'view' causes the utility to open the selected file in a viewer which can display data of the data type held in that file.

5.1 Scenario Architecture

Description of tasks performed by the application is broken down into three discrete steps, for clarity:

- Initialization: The application constructs a view of the file system.
- Getting the Command List: The application presents command list for a selected data item.
- Performing the Command: The application performs a command on the selected data object.

5.2 Initialization

One of the interfaces mentioned below is the 'DataSource' object. Recall that the DataSource object encapsulates the underlying data object in a class that abstracts the underlying data storage mechanism, and presents its consumers with a common data access and typing interface. The file viewer application queries the file system for its contents.

The viewer instantiates a DataSource object for each file in the directory. Then it instantiates a a DataHandler with the DataSource as its constructor argument. A DataHandler can not be instantiated without a DataSource. The DataHandler object provides the client application with access to the CommandMap, which provides a service that enables access to commands that can operate on the data. The application

maintains a list of the DataHandler objects, queries them for their names and icons to generate its display.

```
// for each file in the directory:
File file = new File(file_name);
DataSource ds = new FileDataSource(file);
DataHandler dh = new DataHandler(ds);
```

5.3 Getting the Command List

Once the application has been initialized and has presented a list of files to the user, the user can select a file on the list. When the user selects a file, the application displays a popup menu that lists the available operations on that file.

The application implements this functionality by requesting the list of available commands from the DataHandler object associated with a file. The DataHandler retrieves the MIME Type of the data from the DataSource object and queries the CommandMap for operations that are available on that type. The application interprets the list and presents it to the user on a popup menu. The user then selects one of the operations from that list.

```
// get the command list for an object
CommandInfo cmdInfo[] = dh.getPreferredCommands();

PopupMenu popup = new PopupMenu("Item Menu");

// populate the popup with available commands
for(i = 0; i < cmdInfo.length; i++)
   popup.add(cmdInfo[i].getCommandName());

// add and show popup
add(popup);
popup.show(x_pos, y_pos);
```

5.4 Performing a Command

After the user has selected a command from the popup menu, the application uses the appropriate CommandInfo class to retrieve the JavaBean™ that corresponds to the selected command, and associates the data with that JavaBean™ using the appropriate mechanism (DataHandler, Externalization etc.). Some CommandObjects (viewers for instance) are subclassed from java.awt.Component and require that they are given a parent container. Others (like a default print Command) might not present a user interface. This allows them to be flexible enough to function as stand alone viewer/ editors, or perhaps as components in a compound document system. The 'application' is responsible for providing the proper environment (containment, life cycle, etc.) for the CommandObject to execute in. We expect that the requirements will be lightweight (not much beyond JavaBeans™ containers and AWT containment for visible components).

```
// get the command object
Object cmdBean = cmdInfo[cmd_id].getCommandObject(dh,
                                        this.getClassLoader());
  ... // use serialization/externalization where appropriate
```

```
my_awt_container.add((Component)cmdBean);
```

5.5 An Alternative Scenario

The first scenario was the 'canonical' case. There are also circumstances where the application has already created objects to represent its data. In this case creating an in-memory instance of a DataSource that converted an existing object into an InputStream is an inefficient use of system resources and can result in a loss of data fidelity.

In these cases, the application can instantiate a DataHandler, using the DataHandler(Object obj, String mimeType) constructor. DataHandler implements the Transferable interface, so the consuming JavaBean™ can request representations other than InputStreams. The DataHandler also constructs a DataSource for consumers that request it. The DataContentHandler mechanism is extended to also allow conversion from Objects to InputStreams.

The following code is an example of a data base front end using the JAF, which provides query results in terms of objects.

```
/**
 * Get the viewer to view my query results:
 */
Component getQueryViewer(QueryObject qo) throws Exception {
    String mime_type = qo.getType();
    Object q_result = qo.getResultObject();
    DataHandler my_dh = new DataHandler(q_result, mime_type);

  return
(Component)my_dh.getCommand("view").getCommandObject(my_dh,
null));
  }
```

6.0 Primary Framework Interfaces

This section describes interfaces required to implement the JAF architecture introduced in Section Three.

6.1 The DataSource Interface

The DataSource interface is used by the DataHandler (and possibly other classes elsewhere) to access the underlying data. The DataSource object encapsulates the underlying data object in a class that abstracts the underlying data storage and typing mechanism, and presents its consumers with a common data access interface.

The JAF provides DataSource implementations that support file systems and URLs. Application system vendors can use the DataSource interface to implement their own specialized DataSource classes to support IMAP servers, object databases, or other sources.

There is a one-to-one correspondence between underlying data items (files for instance) and DataSource objects. Also note that the class that implements the DataSource interface is responsible for typing the data. To manage a file system, a DataSource can use a simple mechanism such as a file extension to type data, while a DataSource that supports incoming web-based data can actually examine the data stream to determine its type.

```
public interface DataSource {
 /**
   * Return an InputStream representation of the data
   *
   * This method will throw an exception if it cannot
   * create an InputStream (in cases where DataSource
   * was created with an object and not an input stream)
   *
   * @return an InputStream
   */

 public InputStream getInputStream() throws IOException;

 /**
  * Get the output stream, (write data back to source)
  * @return an OutputStream
  */
 public OutputStream getOutputStream() throws IOException;

 /**
  * Return the base MIME Type of this data. This method
  * is expected to ALWAYS return a valid (non-null)
  * MIME Type. We suggest that if the DataSource implementation
  * cannot determine the type of the data, it use the
  * convention of returning "application/octet-stream".
  * @return the MIME Type
  */
 public String getContentType() throws IOException;

 /**
  * Return the domain specific 'name' of this object. For
  * example, in the case of a file, return the filename.
  */
 public String getName();
```

```
}
```

6.2 The DataHandler Class

The DataHandler class encapsulates a Data object, and provides methods which act on that data.

DataHandler encapsulates the type-to-command object binding service of the Command Map interface for applications. It provides a handle to the operations and data available on a data element.

DataHandler also implements the Transferable interface. This allows applications and applets to retrieve alternative representations of the underlying data, in the form of objects. The DataHandler encapsulates the interface to the component repository and data source.

Let's examine these groups of features in more detail:

6.2.1 Data Encapsulation

A DataHandler object can only be instantiated with data. The data can be in the form of an object implementing the DataSource interface (the preferred way), as an object with an associated content type, or as a URL object.

Once instantiated, the DataHandler tries to provide its data in a flexible way. The DataHandler implements the Transferable interface which allows an object to provide alternative representations of the data. The Transferable interface's functionality can be extended via objects implementing the DataContentHandler interface, and then made available to the DataHandler either by a DataContentHandlerFactory object, or via a CommandMap.

6.2.2 Command Binding

The DataHandler provides wrappers around commonly used functions for command discovery. DataHandler has methods that call into the current CommandMap associated with the DataHandler. By default the DataHandler calls CommandMap's getDefaultCommandMap method if no CommandMap was explicitly set. As a convenience, DataHandler uses the content type of its data when calls are made to the CommandMap.

```
public class DataHandler implements Transferable {
/**
 * DataHandler constructor (DataSource)
 *
 * Initializes the DataHandler class.
 */
public DataHandler(DataSource ds);

/**
 * DataHandler constructor (Object, MIME Type)
 *
 * Initializes the DataHandler class. This constructor
```

```
 * is used when the application already has in memory
 * representations of the data in the form of objects.
 */
public DataHandler(Object obj, String mime_type);

/**
 * Construct a DataHandler from a URL object.
 */
public DataHandler(URL url);

/**
 * Returns the DataSource associated with this
 * instance of DataHandler. In the case of this DataHandler
 * being created using DataHandler(Object, String), the
 * DataHandler will return an instance of DataSource
 * that encapsulates the object.
 */
public DataSource getDataSource() throws IOException;

/**
 * Get the MIME type of the data
 */
public String getContentType();

/**
 * return a data stream for this Object, in DataHandlers
 * created with DataSources and URL will simply return the
 * InputStream, in the Object case, will attempt to find and
 * use a DataContentHandler and use its writeTo method to
 * pipe the objects contents to the InputStream.
 */
public InputStream getInputStream() throws IOException;

/**
 * Get the OutputStream
 *
 * return an output stream for this object, will
 * return null for DataHandler's created with Objects
 */
public OutputStream getOutputStream() throw IOException;

/**
 * Get the name of the data represented by the DataHandler
 * when created with DataSources, with objects it will
 * return null.
 */
public String getName();

/**
 * (from Transferable)
 * Return the MIMETypes (DataFlavor) of this data
 *
 * The return value of this method is derived from the
 * the original type of this data as well as from the
```

```
 * possible Object types returned from a search of
 * the available DataContentHanders.
 * @return the DataFlavors
 */
public synchonized DataFlavor[] getTransferDataFlavors();

/**
 * from Transferable
 */
public boolean isDataFlavorSupported(DataFlavor
                                     flavor);
public Object getTransferData(DataFlavor flavor) throws
                                      IOException,
                                  UnSupportedFlavorException;
/**
 * Set the CommandMap to use, DataHandler uses the
 * CommandMap from the getDefaultCommandMap static
 * method in CommandMap by default.
 */
public void setCommandMap(CommandMap cmdmap);

/**
 * Set the DataContentHandlerFactory for DataHandlers
 */
public static synchronized setDataContentHandlerFactory(
                           DataContentHandlerFactory factory);

/**
 * If the DataHandler was created with a DataSource, the
 * InputStream is retrieved, and the bytes from the InputStream
 * are read to the OutputStream passed in. If the DataHandler
 * was created with an object, the DataContentHandler for the
 * object's type is retrieved and if found, the writeTo
 * method on it is called.
 */
public void writeTo(OutputStream os) throws IOException;

/**
 * Get the content of this DataHandler in terms of an object.
 */
public Object getContent() throws IOException;

/**
 * Get preferred command list.
 *
 * Return an array of CommandInfo classes that describe
 * the preferred (depending on the semantics of the CommandMap)
 * beans that correspond to this objects MIME type. Usually
 * this method will return one CommandInfo for each command for
 * the mimeType. (calls directly into the CommandMap)
 */
CommandInfo[] getPreferredCommands();

/**
```

```
 * Get all the available commands for this type.
 *
 * Return an array of CommandInfo classes that describe
 * all the commands known to the CommandMap that
 * can accept this object's MIME type.
 * (calls directly into the CommandMap)
 */
CommandInfo[] getAllCommands();

/**
 * Get the 'default' command 'cmdName' for this objects
 * MIME type.
 *
 * Attempts to find a command named 'cmdName' that
 * can accept the dh's MIME type. (calls directly into
 * CommandMap)
 */
CommandInfo getCommand(String cmd);

/**
 * Return an instantiated instance of the bean
 *
 * This convenience method uses the CommandInfo class to
 * instantiate an instance of the bean, using this
 * DataHandler instance to set the DataHandler property
 * in beans that implement CommandObject.
 */
Object getBean(CommandInfo binfo);
}
```

6.3 The DataContentHandler Interface

The DataContentHandler interface is used to write objects used by the DataHandler to convert InputStreams into objects. In effect, the DataHandler object uses a DataContentHandler object to implement the Transferable interface. DataContentHandlers are discovered via the current CommandMap. A DataContentHandler uses DataFlavors to represent the data types it can access.

The DataContentHandler also converts data from objects into InputStreams. For instance, if an application needs to access a .gif file, it passes the file to the image/gif DataContentHandler. The .gif DataContentHandler converts the image object into a gif-formatted byte stream.

```
// DataContentHandler
public interface DataContentHandler {
/**
 * return the DataFlavors for this DCH or a zero length array
 */
public DataFlavors[] getTransferDataFlavors();

/**
 * return the Transfer Data
 */
```

```
public Object getTransferData(DataFlavor df, DataSource ds)
throws UnsupportedDataFlavorException,IOException;

/**
 * Return an object representing the content of this DataSource.
 */
public Object getContent(DataSource ds) throws IOException;

/**
 *Write the contents of an object to an OutputStream
 */
public void writeTo(Object obj,
                    String mimeType,
                    OutputStream os)
                                throws IOException;
}
```

6.4 The CommandMap Interface

Once the DataHandler has a MIME Type describing the content, it can query the CommandMap for the operations, or *commands* that are available for that data type. The application requests commands available through the DataHandler and specifies a command on that list. The DataHandler uses the CommandMap to retrieve the JavaBean™ associated with that command. Some or all of the command map is stored in some 'common' place, like a .mailcap (RFC 1524) file. Other more complex implementations can be distributed, or can provide licensing or authentication features.

```
public abstract class CommandMap {
 /**
  * gets the default CommandMap as defined by the implementation
  */
 public static CommandMap getDefaultCommandMap();

 /**
  * sets the DefaultCommandMap
  */
public static void setDefaultCommandMap(CommandMap map);

 /**
  * Get preferred command list.
  *
  * Return a non-null array of CommandInfo objects that describe
  * the preferred (depending on the semantics of the CommandMap)
  * beans that correspond to this mimeType. Usually this
  * method will return one CommandInfo for each command for
  * the mimeType.
  */
 abstract public CommandInfo[] getPreferredCommands(String
                                                    mimeType);
 /**
  * Get all the available commands for this type.
  *
  * Return a non-null array of CommandInfo objects that describe
```

```
     * all the commands known to the CommandMap that
     * can accept this mime type.
     */
    abstract public CommandInfo[] getAllCommands(String mimeType);

    /**
     * Get the 'default' command 'cmdName' for this mime
     * type
     *
     * Attempts to find a command named 'cmdName' that
     * can accept mimeType.
     */
    abstract public CommandInfo getCommand(String mimeType,
                                           String cmdName);
}
```

6.5 The CommandInfo Class

The CommandInfo class is used to represent commands in an underlying registry. From a CommandInfo object, an application can instantiate the JavaBean™ or request the verb (*command*) it describes.

```
public class CommandInfo {
  public CommandInfo(String verb, String className);
  // get the command verb
  public String getCommandName();
  // Return the instantiated object, pass the DataHandler to
  // it if it implements the CommandObject interface. If the
  // DataHandler variable is null, return the instantiated bean.
  public Object getCommandObject(DataHandler dh,
                                 ClassLoader cl);
}
```

6.6 The CommandObject Interface

JavaBeans™ designed specifically for use with the JAF Architecture should implement the CommandObject interface. This interface provides direct access to DataHandler methods and notifies a JAF-aware JavaBean™ which verb was used to call it. Upon instantiation, the JavaBean™ takes a string specifying a user-selected command verb, and the DataHandler object managing the target data. The DataHandler takes a DataSource object, which provides an input stream linked to that data, and a string specifying the data type.

```
public interface CommandObject {
  /**
   * Passes the verb the bean was instantiated from as well
   * as the DataHandler with the bean's data to the bean.
   */
  public void setCommandContext(String verb, DataHandler dh);
}
```

6.7 The DataContentHandlerFactory

Like the ContentHandler factory in the `java.net` package, the DataContentHandlerFactory is an interface that allows developers to write objects that map MIME types to DataContentHandlers. The interface is extremely simple, in order to allow developers as much design and implementation freedom as possible.

```
public interface DataContentHandlerFactory {
    /**
     * create a DataContentHandler for the mimeType
     */
    public DataContentHandler createDataContentHandler(
                                                String mimeType);
}
```

7.0 Writing JavaBeans™ for the Framework

7.1 Overview

This section describes the specification of well-behaved JAF-aware JavaBean™ viewers. Note that this proposal the reader is comfortable with the JavaBeans™ Specification. Developers intending to implement viewer JavaBeans™ for the JAF should be familiar with JavaBean™ concepts and architecture.

7.2 Viewer Goals

1. Make the implementation of viewers and editors as simple as implementing JavaBeans™. i.e.: low cost of entry to be a *good* citizen.
2. Allow developers to have a certain amount of flexibility in their implementations.

7.3 General

We are attempting to limit the amount of extra baggage that needs to be implemented from 'generic' JavaBeans™. In many cases JavaBeans™ which weren't developed with knowledge of the framework can be used. The JAF exploits the existing features of JavaBeans™ and the JDK™, and defines as few additional interfaces and policies as possible.

We expect that in the first release, viewers/editors will be bound to data via a simple registry mechanism similar in function to a .mailcap file. We also plan to exploit any future extensions to the ClassLoader that can allow autodiscovery of configuration files on the system. This would allow developers to deliver supplementary registry files to be appended to the system registry files, allowing additional packages to be added at runtime.

Our viewers/editors and related classes and files are encapsulated into JAR files, as is the preferred method for JavaBeans™. The JAF does not restrict the choice of classes used to implement a JAF-aware 'viewer' JavaBean™, beyond those expected of well-

behaved JavaBeans™. We make no restrictions on which classes are used to implement JavaBeans™ beyond those expected of 'well behaved' JavaBeans™.

7.4 Interfaces

A viewer JavaBean™ that communicates directly with a JAF Datahandler or CommandObject should implement the CommandObject interface. This interface is small and easy to implement. However; JavaBeans™ can still use standard Serialization and Externalization methods available in JDK 1.1 and later versions.

7.5 Storage

The JAF expects applications and viewer JavaBeans™ to implement storage tasks via the DataSource object. However; it is possible to use Externalization. A JAF-aware application can implement the following storage mechanism:

```
ObjectOutputStream oos = new ObjectOutputStream(
                                try {
                                  data_handler.getOutputStream());
                                } catch(IOException e) {}
my_externalizable_bean.writeExternal(oos);
```

7.6 Packaging

The basic format for packaging of the Viewer/Editors is the JAR file as described in the JavaBeans™ Specification. This format allows the convenient packaging of collections of files that are related to a particular JavaBean™ or applet. For more information concerning integration points, see Section 8.

7.7 Container Support

The JAF is designed to be flexible enough to support the needs of a variety of applications. The JAF expects these applications to provide the appropriate containers and life cycle support for these JavaBeans™. JavaBeans™ written for the framework should be compatible with the guidelines in the JavaBeans™ documentation and should be tested against the BDK BeanBox (and the JDK Appletviewer if they are subclassed from Applet).

7.8 Lifecycle

In general the JAF expects that its viewer bean life cycle semantics are the same as those for all JavaBeans™. In the case of JavaBeans™ that implement the CommandObject interface we encourage application developers to not parent JavaBeans™ subclassed from java.awt.Component to an AWT container until after they have set the javax.activation.CommandObject.setCommandContext method.

7.9 Command Verbs

The MailcapCommandMap implementation provides a mechanism that allows for an extensible set of command verbs. Applications using the JAF can query the system for commands available for a particular MIME type, and retrieve the JavaBean™ associated with that MIME type.

8.0 Framework Integration Points

This section presents several examples which clarify how JavaBeans™ developers can write JavaBeans™ that are integrated with the JAF.

First, let's review the pluggable components of the JavaBeans™ Activation Framework:

- A mechanism that accesses target data where it is stored: DataSource
- A mechanism to convert data objects to and from an external byte stream format: DataContentHandler
- A mechanism to locate visual components that operate on data objects: CommandMap
- The visual components that operate on data objects: JAF-aware JavaBeans™

As a JavaBean™ developer, you will build visual JavaBeans™. You can also develop DataContentHandlers to supply data to those JavaBeans™. You might also need to develop a new DataSource or CommandMap class to access data and specify a data type.

8.1 JavaBean™

Suppose you're building a new Wombat Editor product, with its corresponding Wombat file format. You've built the Wombat Editor as one big JavaBean™. Your WombatBean can do anything and everything that you might want to do with a Wombat. It can edit, it can print, it can view, it can save Wombats to files, and it can read Wombats in from files. You've defined a language-independent Wombat file format. You consider the Wombat data and file formats to be proprietary so you have no need to offer programmatic interfaces to Wombats beyond what your WombatBean supports.

You've chosen the MIME type "application/x-wombat" to describe your Wombat file format, and you've chosen the filename extension ".wom" to be used by files containing Wombats.

To integrate with the framework, you'll need some simple wrappers for your WombatBean for each command you want to implement. For example, for a Print command wrapper you can write the following code:

```
public class WombatPrintBean extends WombatBean {
    public WombatPrintBean() {
          super();
          initPrinting();
```

```
        }
    }
```

You will need to create a mailcap file that lists the MIME type “application/x-wombat” and user visible commands that are supported by your WombatBean. Your WombatBean wrappers will be listed as the objects supporting each of these commands.

```
application/x-wombat; ; x-java-view=com.foo.WombatViewBean; \
    x-java-edit=com.foo.WombatEditBean; \
    x-java-print=com.foo.WombatPrintBean
```

You’ll also need to create a mime.types file with an entry:

```
type=application/x-wombat desc="Wombat" exts=wom
```

All of these components are packaged in a JAR file:

```
META-INF/mailcap
META-INF/mime.types
com/foo/WombatBean.class
com/foo/WombetEditBean.class
com/foo/WombatViewBean.class
```

Because everything is built into one JavaBean™, and because no third party programmatic access to your Wombat objects is required, there’s no need for a DataContentHandler. Your WombatBean can therefore implement the Externalizable interface instead; and use its methods to read and write your Wombat files. The DataHandler can call the Externalizable methods when appropriate.

8.2 JavaBeans™

Your Wombat Editor product has really taken off, and you’re now adding significant new functionality and flexibility to your Wombat Editor. It’s no longer feasible to put everything into one giant Bean. Instead, you’ve broken the product into a number of JavaBeans™ and other components:

- A WombatViewer JavaBean™ that can be used to quickly view a Wombat in read-only mode.
- A WombatEditor JavaBean™ that is heavier than the WombatViewer, but also allows editing.
- A WombatPrinter JavaBean™ that simply prints a Wombat.
- A component that reads and writes Wombat files.
- A Wombat class that encapsulates the Wombat data and is used by your other JavaBeans™ and components.

In addition, customers have demanded to be able to programmatically manipulate Wombats, independently from the visual viewer or editor JavaBeans™. You’ll need to create a DataContentHandler that can convert a byte stream to and from a Wombat object. When reading, the WombatDataContentHandler reads a byte stream and returns a new Wombat object. When writing, the WombatDataContentHandler takes a Wombat

object and produces a corresponding byte stream. You'll need to publish the API to the Wombat class.

The WombatDataContentHandler is delivered as a class and is designated as a DataContentHandler that can operate on Wombats in the mailcap file included in JAR file.

Your mailcap file changes to list the appropriate Wombat JavaBeans™, which implement user commands:

```
application/x-wombat; ; x-java-View=com.foo.WombatViewBean; \
          x-java-edit=com.foo.WombatEditBean;    \
          x-java-print=com.foo.WombatPrintBean; \
         x-java-content-handler=com.foo.WombatDataContentHandler
```

Your Wombat JavaBeans™ can continue to implement the Externalizable interface, and thus read and write Wombat byte streams. They are more likely to simply operate on Wombat objects directly. To find the Wombat object they're being invoked to operate on, they implement the CommandObject interface. The setCommandContext method refers them to the corresponding DataHandler, from which they can invoke the getContent method, which will return a Wombat object (produced by the WombatData-ContentHandler).

All components are packaged in a JAR file.

8.3 Viewer Only

The Wombat product has been wildly successful. The ViewAll Company has decided that it can produce a Wombat viewer that's much faster than the WombatViewer JavaBean™. Since they don't want to depend on the presence of any Wombat components, their viewer must parse the Wombat file format, which they reverse engineered.

The ViewAll WombatViewerBean implements the Externalizable interface to read the Wombat data format.

ViewAll delivers an appropriate mailcap file:

```
application/x-wombat; ; x-java-view=com.viewall.WombatViewer
```

and mime.types file:

```
type=application/x-wombat desc="Wombat" exts=wom
```

All components are packaged in a JAR file.

8.4 ContentHandler JavaBean™ Only

Now that everyone is using Wombats, you've decided that it would be nice if you could notify people by email when new Wombats are created. You have designed a new WombatNotification class and a corresponding data format to be sent by email using the

MIME type “application/x-wombat-notification”. Your server detects the presence of new Wombats, constructs a WombatNotification object, and constructs and sends an email message with the Wombat notification data as an attachment. Your customers run a program that scans their email INBOX for messages with Wombat notification attachments and use the WombatNotification class to notify their users of the new Wombats.

In addition to the server application and user application described, you’ll need a DataContentHandler to plug into the DataHandler infrastructure and construct the WombatNotification objects. The WombatNotification DataContentHandler is delivered as a class named WombatNotificationDataContentHandler and is delivered in a JAR file with the following mailcap file:

```
application/x-wombat-notification; \
                              WombatNotificationDataContentHandler
```

The server application creates DataHandlers for its WombatNotification objects. The email system uses the DataHandler to fetch a byte stream corresponding to the WombatNotification object. (The DataHandler uses the DataContentHandler to do this.)

The client application retrieves a DataHandler for the email attachment and uses the getContent method to get the corresponding WombatNotification object, which will then notify the user.

9.0 Framework Deliverables

9.1 Packaging Details

The JAF is implemented as a Standard Extension to the Java™ Platform so it can provide functionality to JDK™ 1.1. This strategy allows the JAF to be delivered asynchronously from the JDK™ and to be included in new software products in a more timely fashion. The following are some more details about the package:

- The package name is `javax.activation`.
- The initial release is supported on JDK™ 1.1.4 and later versions of the JDK™.

9.2 Framework Core Classes

interface DataSource: An interface class that describes a data source which provides a MIME type and an input stream.

class DataHandler: A class that acts as a handle for the data source and uses the existing ContentHandler mechanism and a new similar mechanism to implement the Transferable interface. In addition, it provides access to the registry infrastructure that ‘discovers’ available JAF-aware JavaBeans™.

interface DataContentHandler: An interface similar semantically to the ContentHandler interface that uses DataFlavors and InputStream instead of URLConnections.

interface DataContentHandlerFactory: A factory interface that can be used to implement factories that can be installed into DataHandler.

class CommandMap: An abstract class that describes a registry.

interface CommandObject: An interface that can be implemented by JavaBeans™ that wish to access DataHandlers and the verbs that invoked them directly.

class CommandInfo: A class that is used by CommandMaps to represent JavaBeans™ returned from command requests.

9.3 Framework Auxiliary Classes

class FileDataSource: A simple implementation of a DataSource object that represents a file. This class uses the FileTypeMap mechanism to map files to a MIME type or possibly a .mime.types file. See appendix A.

class FileTypeMap: An abstract class used by the FileDataSource to map files to content types.

class MimetypesFileTypeMap: An implementation of FileTypeMap that uses .mime.types files to map files to content types.

class URLDataSource: A simple implementation of a DataSource object that represents the data pointed to by a URL.

class MailcapCommandMap: A simple sample command map implementation that uses a properties file that is a semantic extension to RFC 1524 (mailcap files) to map MIME types to JavaBeans™. (see appendix A)

class ActivationDataFlavor: Subclassed from `java.awt.datatransfer.DataFlavor`, this implementation of DataFlavor provides support for arbitrary representation classes, and includes more robust MIME type matching.

class UnsupportedDataTypeException: Subclassed from java.io.IOException, it is thrown by the DataHandler when a request is made for an operation requiring a DataContentHandler, but no applicable DataContentHandler is available.

class MimeType: This class provides RFC 2046 MIME Type parsing.

class com.sun.activation.viewers.*: A few simple example viewer JavaBeans™ (text and image).

10.0 Appendix A: Class definitions for default package implementations:

10.1 FileDataSource

```
public class FileDataSource implements DataSource {
  // start with a File
  public FileDataSource(File file);
  // start with a path
  public FileDataSource(String path);

  // return the 'name' of this object
  public String getName();
  // return the content type
  public String getContentType();
  // get the InputStream
  public InputStream getInputStream();
  // get the OutputStream
  public OutputStream getOutputStream();

  // set the FileTypeMap
  public void setFileTypeMap(FileTypeMap);
}
```

10.2 FileTypeMap

```
public abstract class FileTypeMap {
  // Return the type of the file object.
  public String getContentType(File);
  // Return the type of the file
  public String getContentType(String);
  // Return the default FileTypeMap for the system.
  public FileTypeMap getDefaultFileTypeMap();
  // Sets the default FileTypeMap for the system.
  public void setDefaultFileTypeMap(FileTypeMap);
}
```

10.3 MimetypeFileTypeMap

```
public abstract class MimetypesFileTypeMap {
  // The default constructor, reads in the default mimetype
  // mappings included with the JAF.
  public MimetypesFileTypeMap();
  // Constructor that allows one to specify a mimetypes
  // file as an InputStream to append to the registry.
  public MimetypesFileTypeMap(InputStream);
  // Constructor that allows one to specify a mimetypes
  // file as an InputStream to append to the registry.
  public MimetypesFileTypeMap(String);
  // Return the type of the file object.
  public String getContentType(File);
  // Return the type of the file
  public String getContentType(String);
```

```
  // prepends mimetype entries to the registry
  public void addMimeTypes(String);
}
```

10.4 MailcapCommandMap

```
public class MailcapCommandMap extends CommandMap {
  // Default Constructor
  public MailcapCommandMap();
  // Provide a path to a mailcap file
  public MailcapCommandMap(String mailcap);

  // adds mailcap entries to the command map
  public void addMailCap(String);

  // get all know commands for this MIME type
  public CommandInfo[] getAllCommands(String mimeType);

  // get command
  public CommandInfo getCommand(String mimeType,
                                String cmdName);

 // get the preferred set of commands for MIME type
 public CommandInfo[] getPreferredCommands(String mimeType);
}
```

10.5 ActivationDataFlavor

```
public class ActivationDataFlavor extends
                      java.awt.datatransfer.DataFlavor {
 // constructor to construct this DataFlavor with an
 // arbitrary representation class
 public ActivationDataFlavor(Class representationClass,
                             String mimeType,
                             String humanPresentableName);
 // same as super class
 public ActivationDataFlavor(Class representationClass,
                             String humanPresentableName);

// same as super class
 public ActivationDataFlavor(String mimeType,
                             String humanPresentableName);

// same as super class
 public String getMimeType();

 // same as super class
 public Class getRepresentationClass();

 // same as super class
 public void setHumanPresentableName(
```

```
                                    String humanPresentableName);

  // Uses MimeType class to implement more robust parsing
  public boolean equals(DataFlavor dataFlavor);

  // Uses MimeType class to implement more robust parsing
  public boolean isMimeTypeEqual(String mimeType);

  // same as super class
  protected String normalizeMimeTypeParameter(
                                          String parameterName,
                                          String parameterValue);

  // same as super class
  protected String normalizeMimeType(String mimeType);
 }
```

11.0 Document Change History

May 13,1997:Initial Public Draft 1

Aug 1, 1997: Internal Review Draft 2

- Added *Integration Points* section
- Minor API changes

Sept 16 1997:Second Public Draft 3

- Edited document to reflect change to Standard Extension
- Removed URL/URLConnection section
- Minor API changes

Oct 28 1997: Third Public Draft 4

- Minor API changes
- Add additional class descriptions
- Fixed minor errata

Dec 9, 1997: Fourth Public Draft 5

- Minor API changes
- Add additional class descriptions
- Fixed minor errata
- Includes **Frozen** API

Feb. 20, 1998: Version 0.6

- Minor typos fixed.
- Change bars removed.

Mar. 16, 1998: Version 1.0

- Version 1.0

12.0 Contacting Us

Please send your questions and comments to:

`activation-comments@icdev.eng.sun.com`