JavaBeans™ Activation Framework Specification *Version 1.0a*

Bart Calder, Bill Shannon

This is the JavaBeans™ Activation Framework Specification, a data typing and registry technology that is 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™ was 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:

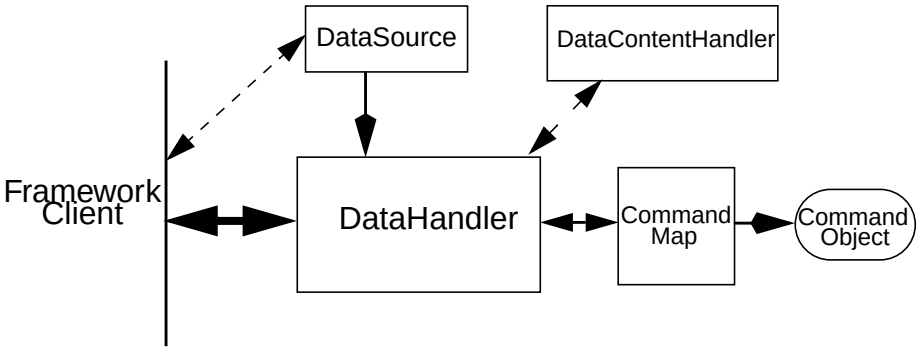
- It determines the type of arbitrary data.
- It encapsulates access to data.
- It discovers the operations available on a particular type of data.

- 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 JavaBeans™ component 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

Beans extend the CommandObject interface in order to interact with JAF services. JAF-aware JavaBeans™ components 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 Beans) 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 Bean which is used to operate on the data. The container (user of the framework) makes the association between the data and the Bean.

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 the 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 Bean that corresponds to the selected command, and associates the data with that Bean 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 Bean 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 {
 /**
   * This methodreturns an <code>InputStream</code>
   * representing the data and throws the appropriate
   * exception if it can not do so.
   *
   * @return an InputStream
   */
  public InputStream getInputStream() throws IOException;

 /**
  * This method returns an <code>OutputStream</code> where
  * the data can be written and throws the appropriate exception
  * if it can not do so.
  *
  * @return an OutputStream
  */
 public OutputStream getOutputStream() throws IOException;

 /**
  * This method returns the MIME Type of the data in the form
  * of a string. It should always return a valid type.  It is
  * suggested that getContentType return
  * "application/octet-stream" if the DataSource implementation
  * can not determine the data type.
  *
  * @return the MIME Type
  */
 public String getContentType();

 /**
  * Return the <i>name</i> of this object where the name of the
  * object is dependant on the nature of the underlying objects.
  * DataSources encapsulating files may choose to return the
  * filename of the object.  (Typically this would be the last
  * component of the filename, not an entire pathname.)
  *
  * @return the name of the object.
  */
 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 {
/**
 * Create a <code>DataHandler</code> instance referencing the
 * specified DataSource.  The data exists in a byte stream form.
 * The DataSource will provide an InputStream to access the
 * data.
 *
 * @param ds        the DataSource
 */
public DataHandler(DataSource ds);

/**
```

```
     * Create a <code>DataHandler</code> instance representing an
     * object of this MIME type.  This constructor is
     * used when the application already has an in-memory
     * representation of the data in the form of a Java Object.
     *
     * @param obj       the Java Object
     * @param mimeType  the MIME type of the object
     */
    public DataHandler(Object obj, String mime_type);

    /**
     * Create a <code>DataHandler</code> instance referencing a URL.
     * The DataHandler internally creates a
     * <code>URLDataSource</code> instance to represent the URL.
     *
     * @param url       a URL object
     */
    public DataHandler(URL url);

    /**
     * Return the DataSource associated with this
     * instance of DataHandler.
     * <p>
     * For DataHandlers that have been instantiated with a
     * DataSource, this method returns the DataSource that was used
     * to create the DataHandler object. In other cases the
     * DataHandler constructs a DataSource from the data used to
     * construct the DataHandler. DataSources created for
     * DataHandlers <b>not</b> instantiated with a DataSource are
     * cached for performance reasons.
     *
     * @return  a valid DataSource object for this DataHandler
     */
    public DataSource getDataSource();

    /**
     * Return the name of the data object. If this DataHandler
     * was created with a DataSource, this method calls through
     * to the <code>DataSource.getName</code> method, otherwise it
     * returns <i>null</i>.
     *
     * @return  the name of the object
     */
    public String getName();

    /**
     * Return the MIME type of this object as retrieved from
     * the source object. Note that this is the <i>full</i>
     * type with parameters.
     *
     * @return  the MIME type
     */
    public String getContentType();
```

```
/**
 * Get the InputStream for this object. <p>
 *
 * For DataHandlers instantiated with a DataSource, the
 * DataHandler calls the <code>DataSource.getInputStream</code>
 * method and returns the result to the caller.
 * <p>
 * For DataHandlers instantiated with an Object, the DataHandler
 * first attempts to find a DataContentHandler for the Object.
 * If the DataHandler can not find a DataContentHandler for this
 * MIME type, it throws an UnsupportedDataTypeException.  If it
 * is successful, it creates a pipe and a thread.  The thread
 * uses the DataContentHandler's <code>writeTo</code> method to
 * write the stream data into one end of the pipe.  The other
 * end of the pipe is returned to the caller.  Because a thread
 * is created to copy the data, IOExceptions that may occur
 * during the copy can not be propagated back to the caller. The
 * result is an empty stream.<p>
 *
 * @return  the InputStream representing this data
 * @exception IOException   if an I/O error occurs
 *
 * @see javax.activation.DataContentHandler#writeTo
 * @see javax.activation.UnsupportedDataTypeException
 */
public InputStream getInputStream() throws IOException;

/**
 * Get an OutputStream for this DataHandler to allow overwriting
 * the underlying data.
 * If the DataHandler was created with a DataSource, the
 * DataSource's <code>getOutputStream</code> method is called.
 * Otherwise, <code>null</code> is returned.
 *
 * @return the OutputStream
 *
 * @see javax.activation.DataSource#getOutputStream
 * @see javax.activation.URLDataSource
 */
public OutputStream getOutputStream() throw IOException;

/**
 * Return the DataFlavors in which this data is available. <p>
 *
 * Returns an array of DataFlavor objects indicating the flavors
 * the data can be provided in. The array is usually ordered
 * according to preference for providing the data, from most
 * richly descriptive to least richly descriptive.<p>
 *
 * The DataHandler attempts to find a DataContentHandler that
 * corresponds to the MIME type of the data. If one is located,
 * the DataHandler calls the DataContentHandler's
 * <code>getTransferDataFlavors</code> method. <p>
 *
```

```
     * If a DataContentHandler can <i>not</i> be located, and if the
     * DataHandler was created with a DataSource (or URL), one
     * DataFlavor is returned that represents this object's MIME
     * type and the <code>java.io.InputStream</code> class.  If the
     * DataHandler was created with an object and a MIME type,
     * getTransferDataFlavors returns one DataFlavor that represents
     * this object's MIME type and the object's class.
     *
     * @return  an array of data flavors in which this data can be
     *          transferred
     *
     * @see
     * javax.activation.DataContentHandler#getTransferDataFlavors
     */
    public synchonized DataFlavor[] getTransferDataFlavors();

    /**
     * Returns whether the specified data flavor is supported
     * for this object.<p>
     *
     * This method iterates through the DataFlavors returned from
     * <code>getTransferDataFlavors</code>, comparing each with
     * the specified flavor.
     *
     * @param flavor    the requested flavor for the data
     * @return          true if the data flavor is supported
     *
     * @see javax.activation.DataHandler#getTransferDataFlavors
     */
    public boolean isDataFlavorSupported(DataFlavor flavor);

    /**
     * Returns an object that represents the data to be
     * transferred. The class of the object returned is defined by
     * the representation class of the data flavor.<p>
     *
     * <b>For DataHandler's created with DataSources or URLs:</b><p>
     *
     * The DataHandler attempts to locate a DataContentHandler
     * for this MIME type. If one is found, the passed in DataFlavor
     * and the type of the data are passed to its
     * <code>getTransferData</code> method. If the DataHandler fails
     * to locate a DataContentHandler and the flavor specifies this
     * object's MIME type and the <code>java.io.InputStream</code>
     * class, this object's InputStream is returned.
     * Otherwise it throws an UnsupportedFlavorException. <p>
     *
     * <b>For DataHandler's created with Objects:</b><p>
     *
     * The DataHandler attempts to locate a DataContentHandler
     * for this MIME type. If one is found, the passed in DataFlavor
     * and the type of the data are passed to its getTransferData
     * method. If the DataHandler fails to locate a
     * DataContentHandler and the flavor specifies this object's
```

```
 * MIME type and its class, this DataHandler's referenced object
 * is returned. Otherwise it throws an
 * UnsupportedFlavorException.
 *
 * @param flavor    the requested flavor for the data
 * @return          the object
 * @exception UnsupportedFlavorException    if the data could
 *                  not be converted to the requested flavor
 * @exception IOException   if an I/O error occurs
 *
 * @see javax.activation.ActivationDataFlavor
 */
public Object getTransferData(DataFlavor flavor) throws
                  UnSupportedFlavorException, IOException;

/**
 * Set the CommandMap for use by this DataHandler.
 * Setting it to <code>null</code> causes the CommandMap to
 * revert to the CommandMap returned by the
 * <code>CommandMap.getDefaultCommandMap</code> method.
 * Changing the CommandMap, or setting it to <code>null</code>,
 * clears out any data cached from the previous CommandMap.
 *
 * @param commandMap        the CommandMap to use in this
 *                          DataHandler
 *
 * @see javax.activation.CommandMap#setDefaultCommandMap
 */
public synchronized void setCommandMap(CommandMap commandmap);

/**
 * Sets the DataContentHandlerFactory.  The
 * DataContentHandlerFactory is called first to find
 * DataContentHandlers.  The DataContentHandlerFactory can only
 * be set once.
 * <p>
 * If the DataContentHandlerFactory has already been set,
 * this method throws an Error.
 *
 * @param factory   the DataContentHandlerFactory
 * @exception Error if the factory has already been defined.
 *
 * @see javax.activation.DataContentHandlerFactory
 */
public static synchronized void setDataContentHandlerFactory(
                        DataContentHandlerFactory newFactory);

/**
 * Write the data to an <code>OutputStream</code>.<p>
 *
 * If the DataHandler was created with a DataSource, writeTo
 * retrieves the InputStream and copies the bytes from the
 * InputStream to the OutputStream passed in.
 * <p>
```

```
     * If the DataHandler was created with an object, writeTo
     * retrieves the DataContentHandler for the object's type.
     * If the DataContentHandler was found, it calls the
     * <code>writeTo</code> method on the
     * <code>DataContentHandler</code>.
     *
     * @param os        the OutputStream to write to
     * @exception IOException   if an I/O error occurs
     */
    public void writeTo(OutputStream os) throws IOException;

    /**
     * Return the data in its preferred Object form. <p>
     *
     * If the DataHandler was instantiated with an object, return
     * the object. <p>
     *
     * If the DataHandler was instantiated with a DataSource,
     * this method uses a DataContentHandler to return the content
     * object for the data represented by this DataHandler. If no
     * <code>DataContentHandler</code> can be found for the
     * the type of this data, the DataHandler returns an
     * InputStream for the data.
     *
     * @return the content.
     * @exception IOException if an IOException occurs during
     *                              this operation.
     */
    public Object getContent() throws IOException;

    /**
     * Return the <i>preferred</i> commands for this type of data.
     * This method calls the <code>getPreferredCommands</code>
     * method in the CommandMap associated with this instance of
     * DataHandler.  This method returns an array that represents a
     * subset of available commands. In cases where multiple
     * commands for the MIME type represented by this DataHandler
     * are present, the
     * installed CommandMap chooses the appropriate commands.
     *
     * @return  the CommandInfo objects representing the preferred
     * commands
     *
     * @see javax.activation.CommandMap#getPreferredCommands
     */
    public CommandInfo[] getPreferredCommands();

    /**
     * Return all the commands for this type of data.
     * This method returns an array containing all commands
     * for the type of data represented by this DataHandler. The
     * MIME type for the underlying data represented by this
     * DataHandler is used to call through to the
     * <code>getAllCommands</code> method of the CommandMap
```

```
     * associated with this DataHandler.
     *
     * @return  the CommandInfo objects representing all the
     * commands
     *
     * @see javax.activation.CommandMap#getAllCommands
     */
    public CommandInfo[] getAllCommands();

    /**
     * Get the command <i>cmdName</i>. Use the search semantics as
     * defined by the CommandMap installed in this DataHandler. The
     * MIME type for the underlying data represented by this
     * DataHandler is used to call through to the
     * <code>getCommand</code> method of the CommandMap associated
     * with this DataHandler.
     *
     * @param cmdName    the command name
     * @return  the CommandInfo corresponding to the command
     *
     * @see javax.activation.CommandMap#getCommand
     */
    public CommandInfo getCommand(String cmdName);

    /**
     * A convenience method that takes a CommandInfo object
     * and instantiates the corresponding command, usually
     * a JavaBeans component.
     * <p>
     * This method calls the CommandInfo's
     * <code>getCommandObject</code>
     * method with the <code>ClassLoader</code> used to load
     * the <code>javax.activation.DataHandler</code> class itself.
     *
     * @param cmdinfo the CommandInfo corresponding to a command
     * @return  the instantiated command object
     */
    public Object getBean(CommandInfo cmdinfo);
    }
```

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.

```
public interface DataContentHandler {
/**
 * Returns an array of DataFlavor objects indicating the flavors
 * the data can be provided in. The array should be ordered
 * according to preference for providing the data (from most
 * richly descriptive to least descriptive).
 *
 * @return The DataFlavors.
 */
public DataFlavor[] getTransferDataFlavors();

/**
 * Returns an object which represents the data to be
 * transferred. The class of the object returned is defined by
 * the representation class of the flavor.
 *
 * @param df The DataFlavor representing the requested type.
 * @param ds The DataSource representing the data to be
 *           converted.
 * @return The constructed Object.
 */
public Object getTransferData(DataFlavor df, DataSource ds)
             throws UnsupportedDataFlavorException,IOException;

/**
 * Return an object representing the data in its most preferred
 * form. Generally this will be the form described by the first
 * DataFlavor returned by the
 * <code>getTransferDataFlavors</code> method.
 *
 * @param ds The DataSource representing the data to be
 *           converted.
 * @return The constructed Object.
 */
public Object getContent(DataSource ds) throws IOException;

/**
 * Convert the object to a byte stream of the specified MIME
 * type and write it to the output stream.
 *
 * @param obj      The object to be converted.
 * @param mimeType The requested MIME type of the resulting byte
 *                 stream.
 * @param os       The output stream into which to write the
 *                 converted byte stream.
 */
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 Bean 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 {
 /**
 * Get the default CommandMap.
 * <li> In cases where a CommandMap instance has been previously
 * set to some value (via <i>setDefaultCommandMap</i>)
 *  return the CommandMap.
 * <li>
 *  In cases where no CommandMap has been set, the CommandMap
 *  creates an instance of <code>MailcapCommandMap</code> and
 * set that to the default, returning its value.
 *
 * @return the CommandMap
 */
 public static CommandMap getDefaultCommandMap();

/**
 * Set the default CommandMap. Reset the CommandMap to the
 * default by calling this method with <code>null</code>.
 *
 * @param commandMap The new default CommandMap.
 * @exception SecurityException if the caller doesn't have
 *            permission to change the default
 */
public static void setDefaultCommandMap(CommandMap commandMap);

/**
 * Get the preferred command list from a MIME Type. The actual
 * semantics
 * are determined by the implementation of the CommandMap.
 *
 * @param mimeType    the MIME type
 * @return the CommandInfo classes that represent the command
 *         Beans.
 */
 abstract public CommandInfo[] getPreferredCommands(String
                                                    mimeType);
/**
 * Get all the available commands for this type. This method
 * should return all the possible commands for this MIME type.
 *
 * @param mimeType    the MIME type
 * @return the CommandInfo objects representing all the
 *         commands.
 */
```

```
 abstract public CommandInfo[] getAllCommands(String mimeType);

/**
 * Get the default command corresponding to the MIME type.
 *
 * @param mimeType    the MIME type
 * @param cmdName     the command name
 * @return the CommandInfo corresponding to the command.
 */
abstract public CommandInfo getCommand(String mimeType,
                                       String cmdName);

/**
 * Locate a DataContentHandler that corresponds to the MIME
 * type.  The mechanism and semantics for determining this are
 * determined by the implementation of the particular
 * CommandMap.
 *
 * @param mimeType        the MIME type
 * @return the DataContentHandler for the MIME type
 */
abstract public DataContentHandler
               createDataContentHandler(String mimeType);
}
```

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 Bean or request the verb (*command*) it describes.

```
public class CommandInfo {
/**
 * The Constructor for CommandInfo.
 * @param verb The command verb this CommandInfo decribes.
 * @param className The command's fully qualified class name.
 */
  public CommandInfo(String verb, String className);

/**
 * Return the command verb.
 *
 * @return the command verb.
 */
  public String getCommandName();

/**
 * Return the command's class name. <i>This method MAY return
 * null in cases where a CommandMap subclassed CommandInfo for
 * its own purposes.</i> In other words, it might not be
 * possible to create the correct state in the command by merely
 * knowing its class name. <b>DO NOT DEPEND ON THIS METHOD
 * RETURNING A VALID VALUE!</b>
 *
```

```
     * @return The class name of the command, or <i>null</i>
     */
      public String getCommandClass() {
            return className;
      }

    /**
     * Return the instantiated JavaBeans component.
     * <p>
     * Begin by instantiating the component with
     * <code>Beans.instantiate()</code>.
     * <p>
     * If the bean implements the
     * <code>javax.activation.CommandObject</code> interface, call
     * its <code>setCommandContext</code> method.
     * <p>
     * If the DataHandler parameter is null, then the bean is
     * instantiated with no data. NOTE: this may be useful
     * if for some reason the DataHandler that is passed in
     * throws IOExceptions when this method attempts to
     * access its InputStream. It will allow the caller to
     * retrieve a reference to the bean if it can be
     * instantiated.
     * <p>
     * If the bean does NOT implement the CommandObject interface,
     * this method will check if it implements the
     * java.io.Externalizable interface. If it does, the bean's
     * readExternal method will be called if an InputStream
     * can be acquired from the DataHandler.<p>
     *
     * @param dh       The DataHandler that describes the data to be
     *                  passed to the command.
     * @param loader    The ClassLoader to be used to instantiate
     *                  the bean.
     * @return The bean
     *
     * @see java.beans.Beans#instantiate
     * @see javax.activation.CommandObject
     */
      public Object getCommandObject(DataHandler dh,
                                     ClassLoader cl)
                    throws IOException, ClassNotFoundException;
    }
```

6.6 The CommandObject Interface

Beans 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 Bean which verb was used to call it. Upon instantiation, the Bean 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 {
/**
 * Initialize the Command with the verb it is requested to
 * handle and the DataHandler that describes the data it will
 * operate on. <b>NOTE:</b> it is acceptable for the caller
 * to pass <i>null</i> as the value for
 * <code>DataHandler</code>.
 *
 * @param verb The Command Verb this object refers to.
 * @param dh The DataHandler.
 */
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 {
/**
 * Creates a new DataContentHandler object for the MIME type.
 *
 * @param mimeType the MIME type to create the
 * DataContentHandler for.
 * @return The new <code>DataContentHandler</code>,
 * or <i>null</i> if none are found.
 */
public DataContentHandler createDataContentHandler(
                                                String mimeType);
}
```

7.0 Writing Beans for the Framework

7.1 Overview

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

7.2 Viewer Goals

1. Make the implementation of viewers and editors as simple as implementing Beans. 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’ Beans. In many cases JavaBeans™ components 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’ Beans, beyond those expected of well-behaved Beans. We make no restrictions on which classes are used to implement Beans beyond those expected of ‘well behaved’ JavaBeans™ components.

7.4 Interfaces

A viewer Bean that communicates directly with a JAF Datahandler or CommandObject should implement the CommandObject interface. This interface is small and easy to implement. However; Beans 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 Beans 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 Bean 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 Beans. Beans 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 Beans. In the case of Beans that implement the CommandObject interface we encourage application developers to not parent Beans 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 Bean associated with that MIME type.

8.0 Framework Integration Points

This section presents several examples which clarify how JavaBeans™ developers can write Beans 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 Beans

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

8.1 Bean

Suppose you're building a new Wombat Editor product, with its corresponding Wombat file format. You've built the Wombat Editor as one big Bean. 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 Bean, 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 Beans

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 Beans and other components:

- A WombatViewer Bean that can be used to quickly view a Wombat in read-only mode.
- A WombatEditor Bean that is heavier than the WombatViewer, but also allows editing.
- A WombatPrinter Bean 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 Beans and components.

In addition, customers have demanded to be able to programmatically manipulate Wombats, independently from the visual viewer or editor Beans. 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 Beans, 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 Beans 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 Bean. 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 Bean 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: The DataSource interface provides the JavaBeans Activation Framework with an abstraction of some arbitrary collection of data. It provides a type for that data as well as access to it in the form of InputStreams and OutputStreams where appropriate.

class DataHandler: The DataHandler class provides a consistent interface to data available in many different sources and formats. It manages simple stream to string conversions and related operations using DataContentHandlers. It provides access to commands that can operate on the data. The commands are found using a CommandMap.

interface DataContentHandler: The DataContentHandler interface is implemented by objects that can be used to extend the capabilities of the DataHandler's implementation of the Transferable interface. Through DataContentHandlers the framework can be extended to convert streams in to objects, and to write objects to streams.

interface DataContentHandlerFactory: This interface defines a factory for DataContentHandlers. An implementation of this interface should map a MIME type into an instance of DataContentHandler. The design pattern for classes implementing this interface is the same as for the ContentHandler mechanism used in java.net.URL.

class CommandMap: The CommandMap class provides an interface to the registry of viewer, editor, print, etc. objects available in the system. Developers are expected to either use the CommandMap implementation included with this package (MailcapCommandMap) or develop their own. Note that some of the methods in this class are abstract.

interface CommandObject: Beans that are Activation Framework aware implement this interface to find out which command verb they're being asked to perform, and to obtain the DataHandler representing the data they should operate on. Beans that don't implement this interface may be used as well. Such commands may obtain the data using the Externalizable interface, or using an application-specific method.

class CommandInfo: The CommandInfo class is used by CommandMap implementations to describe the results of command requests. It provides the requestor with both the verb requested, as well as an instance of the bean. There is also a method that will return the name of the class that implements the command but it is not guaranteed to return a valid value. The reason for this is to allow CommandMap implmentations that subclass CommandInfo to provide special behavior. For example a

CommandMap could dynamically generate Beans. In this case, it might not be possible to create an object with all the correct state information solely from the class name.

9.3 Framework Auxiliary Classes

class FileDataSource: The FileDataSource class implements a simple DataSource object that encapsulates a file. It provides data typing services via a FileTypeMap object. (See appendix A.)

class FileTypeMap: The FileTypeMap is an abstract class that provides a data typing interface for files. Implementations of this class will implement the getContentType methods which will derive a content type from a file name or a File object. FileTypeMaps could use any scheme to determine the data type, from examining the file extension of a file (like the MimetypesFileTypeMap) to opening the file and trying to derive its type from the contents of the file. The FileDataSource class uses the default FileTypeMap (a MimetypesFileTypeMap unless changed) to determine the content type of files.

class MimetypesFileTypeMap: This class extends FileTypeMap and provides data typing of files via their file extension. It uses the .mime.types format.

class URLDataSource: The URLDataSource class provides an object that wraps a URL object in a DataSource interface. URLDataSource simplifies the handling of data described by URLs within the JavaBeans Activation Framework because this class can be used to create new DataHandlers.

class MailcapCommandMap: MailcapCommandMap extends the CommandMap abstract class. It implements a CommandMap whose configuration is based on mailcap files (RFC 1524). The MailcapCommandMap can be configured both programmatically and via configuration files. (See appendix A.)

class ActivationDataFlavor: The ActivationDataFlavor is a special subclass of java.awt.datatransfer.DataFlavor. It allows the JAF to set all three values stored by the DataFlavor class via a new constructor as well as improved MIME parsing in the equals method. Except for the improved parsing, its semantics are identical to that of the JDK's DataFlavor class.

class UnsupportedDataTypeException: Signals that requested operation does not support the requested data type.

class MimeType: A Multipurpose Internet Extension (MIME) type, as defined in RFC 2045 and 2046.

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

10.0 Appendix A: Class definitions for default package implementations:

10.1 FileDataSource

```
public class FileDataSource implements DataSource {
/**
 * Creates a FileDataSource from a File object. <i>Note:
 * The file will not actually be opened until a method is
 * called that requires the file to be opened.</i>
 *
 * @param file the file
 */
  public FileDataSource(File file);

/**
 * Creates a FileDataSource from the specified path name.
 * <i>Note:  The file will not actually be opened until a method
 * is called that requires the file to be opened.</i>
 *
 * @param name the system-dependent file name.
 */
  public FileDataSource(String name);

/**
 * Return the <i>name</i> of this object. The FileDataSource
 * will return the file name of the object.
 *
 * @return the name of the object.
 *
 * @see javax.activation.DataSource
 */
  public String getName();

/**
 * This method returns the MIME type of the data in the form of
 * a string. This method uses the currently installed
 * FileTypeMap. If there is no FileTypeMap explictly set, the
 * FileDataSource will call the
 * <code>getDefaultFileTypeMap</code> method on FileTypeMap to
 * acquire a default FileTypeMap. <i>Note: By default, the
 * FileTypeMap used will be a MimetypesFileTypeMap.</i>
 *
 * @return the MIME Type
 *
 * @see javax.activation.FileTypeMap#getDefaultFileTypeMap
 */
  public String getContentType();

/**
 * This method will return an InputStream representing the
 * the data and will throw an IOException if it can
 * not do so. This method will return a new
```

```
     * instance of InputStream with each invocation.
     *
     * @return an InputStream
     */
      public InputStream getInputStream() throws IOException;

    /**
     * This method will return an OutputStream representing the
     * the data and will throw an IOException if it can
     * not do so. This method will return a new instance of
     * OutputStream with each invocation.
     *
     * @return an OutputStream
     */
      public OutputStream getOutputStream() throws IOException;

    /**
     * Return the File object that corresponds to this
     * FileDataSource.
     * @return the File object for the file represented by this
     * object.
     */
      public File getFile();

    /**
     * Set the FileTypeMap to use with this FileDataSource
     *
     * @param map The FileTypeMap for this object.
     */
      public void setFileTypeMap(FileTypeMap map);
    }
```

10.2 FileTypeMap

```
    public abstract class FileTypeMap {
    /**
     * The default constructor.
     */
      public FileTypeMap();

    /**
     * Return the type of the file object. This method should
     * always return a valid MIME type.
     *
     * @param f A file to be typed.
     * @return The content type.
     */
      abstract public String getContentType(File file);

    /**
     * Return the type of the file passed in.  This method should
     * always return a valid MIME type.
     *
     * @param filename the pathname of the file.
```

```
     * @return The content type.
     */
      abstract public String getContentType(String filename);

    /**
     * Return the default FileTypeMap for the system.
     * If setDefaultFileTypeMap was called, return
     * that instance, otherwise return an instance of
     * <code>MimetypesFileTypeMap</code>.
     *
     * @return The default FileTypeMap
     *
     * @see javax.activation.FileTypeMap#setDefaultFileTypeMap
     */
      public static FileTypeMap getDefaultFileTypeMap();

    /**
     * Sets the default FileTypeMap for the system. This instance
     * will be returned to callers of getDefaultFileTypeMap.
     *
     * @param map The FileTypeMap.
     * @exception SecurityException if the caller doesn't have
     * permission to change the default
     */
      public static void setDefaultFileTypeMap(FileTypeMap map);
    }
```

10.3 MimetypesFileTypeMap

```
    public abstract class MimetypesFileTypeMap {
    /**
     * The default constructor.
     */
      public MimetypesFileTypeMap();

    /**
     * Construct a MimetypesFileTypeMap with programmatic entries
     * added from the InputStream.
     *
     * @param is      the input stream to read from
     */
      public MimetypesFileTypeMap(InputStream is);

    /**
     * Construct a MimetypesFileTypeMap with programmatic entries
     * added from the named file.
     *
     * @param mimeTypeFileName        the file name
     */
      public MimetypesFileTypeMap(String mimeTypeFileName)
                                  throws IOException;

    /**
     * Return the MIME type of the file object.
```

```
 * The implementation in this class calls
 * <code>getContentType(f.getName())</code>.
 *
 * @param f       the file
 * @return        the file's MIME type
 */
  public String getContentType(File f);

/**
 * Return the MIME type based on the specified file name.
 * The MIME type entries are searched as described above under
 * <i>MIME types file search order</i>.
 * If no entry is found, the type "application/octet-stream" is
 * returned.
 *
 * @param filename        the file name
 * @return                the file's MIME type
 */
  public synchronized String getContentType(String filename);

/**
 * Prepend the MIME type values to the registry.
 *
 * @param mime_types  A .mime.types formatted string of entries.
 */
  public synchronized void addMimeTypes(String mime_types);
}
```

10.4 MailcapCommandMap

```
public class MailcapCommandMap extends CommandMap {
/**
 * The default Constructor.
 */
  public MailcapCommandMap();

/**
 * Constructor that allows the caller to specify the path
 * of a <i>mailcap</i> file.
 *
 * @param fileName The name of the <i>mailcap</i> file to open
 */
  public MailcapCommandMap(String filename) throws IOException;

/**
 * Constructor that allows the caller to specify an
 * <i>InputStream</i> containing a mailcap file.
 *
 * @param is        InputStream of the <i>mailcap</i> file to
 * open
 */
  public MailcapCommandMap(InputStream is);

/**
```

```
     * Add entries to the registry.  Programmatically
     * added entries are searched before other entries.<p>
     *
     * The string that is passed in should be in mailcap
     * format.
     *
     * @param mail_cap a correctly formatted mailcap string
     */
      public synchronized void addMailCap(String mail_cap);

    /**
     * Get all the available commands in all mailcap files known to
     * this instance of MailcapCommandMap for this MIME type.
     *
     * @return the CommandInfo objects representing all the
     * commands.
     */
      public synchronized CommandInfo[] getAllCommands(String
                                                       mimeType);

    /**
     * Get the command corresponding to <code>cmdName</code> for the
     * MIME type.
     *
     * @return the CommandInfo object corresponding to the command.
     */
      public synchronized CommandInfo getCommand(String mimeType,
                                    String cmdName);

    /**
     * Get the preferred command list for a MIME Type. The
     * MailcapCommandMap searches the mailcap files as described
     * above under <i>Mailcap file search order</i>.<p>
     *
     * The result of the search is a proper subset of available
     * commands in all mailcap files known to this instance of
     * MailcapCommandMap.  The first entry for a particular command
     * is considered the preferred command.
     *
     * @return the CommandInfo objects representing the preferred
     * commands.
     */
     public synchronized CommandInfo[] getPreferredCommands(String
                                                         mimeType);

    /**
     * Return the DataContentHandler for the specified MIME type.
     */
     public synchronized DataContentHandler
            createDataContentHandler(String mimeType);
    }
```

10.5 ActivationDataFlavor

```
public class ActivationDataFlavor extends
                      java.awt.datatransfer.DataFlavor {

/**
 * Construct a DataFlavor that represents an arbitrary
 * Java object. This constructor is an extension of the
 * JDK's DataFlavor in that it allows the explicit setting
 * of all three DataFlavor attributes.
 * <p>
 * The returned DataFlavor will have the following
 * characteristics:
 * <p>
 * representationClass = representationClass<br>
 * mimeType            = mimeType<br>
 * humanName           = humanName
 * <p>
 *
 * @param representationClass the class used in this DataFlavor
 * @param mimeType the MIME type of the data represented by this
 * class
 * @param humanPresentableName the human presentable name of the
 * flavor
*/
 public ActivationDataFlavor(Class representationClass,
                             String mimeType,
                             String humanPresentableName);

/**
 * Construct a DataFlavor that represents a MimeType.
 * <p>
 * The returned DataFlavor will have the following
 * characteristics:
 * <p>
 * If the mimeType is "application/x-java-serialized-object;
 * class=", the result is the same as calling new
 * DataFlavor(Class.forName()) as above.
 * <p>
 * otherwise:
 * <p>
 * representationClass = InputStream<p>
 * mimeType = mimeType<p>
 *
 * @param representationClass the class used in this DataFlavor
 * @param humanPresentableName the human presentable name of the
 *                              flavor
 */
 public ActivationDataFlavor(Class representationClass,
                             String humanPresentableName);

/**
 * Construct a DataFlavor that represents a MimeType.
 * <p>
```

```
     * The returned DataFlavor will have the following
     * characteristics:
     * <p>
     * If the mimeType is "application/x-java-serialized-object;
     * class=", the result is the same as calling new
     * DataFlavor(Class.forName()) as above, otherwise:
     * <p>
     * representationClass = InputStream<p>
     * mimeType = mimeType
     *
     * @param mimeType the MIME type of the data represented by this
     *                 class
     * @param humanPresentableName the human presentable name of the
     *                             flavor
     */
     public ActivationDataFlavor(String mimeType,
                                 String humanPresentableName);

    /**
     * Return the MIME type for this DataFlavor.
     *
     * @return    the MIME type
     */
     public String getMimeType();

    /**
     * Return the representation class.
     *
     * @return the representation class
     */
     public Class getRepresentationClass();

    /**
     * Return the Human Presentable name.
     *
     * @return     the human presentable name
     */
     public String getHumanPresentableName();

    /**
     * Set the human presentable name.
     *
     * @param humanPresentableName    the name to set
     */
     public void setHumanPresentableName(
                                 String humanPresentableName);

    /**
     * Compares the DataFlavor passed in with this DataFlavor; calls
     * the <code>isMimeTypeEqual</code> method.
     *
     * @param dataFlavor      the DataFlavor to compare with
     * @return                true if the MIME type and
     *                        representation class are the same
```

```
 */
 public boolean equals(DataFlavor dataFlavor);

/**
 * Is the string representation of the MIME type passed in
 * equivalent to the MIME type of this DataFlavor. <p>
 *
 * ActivationDataFlavor delegates the comparison of MIME types
 * to the MimeType class included as part of the JavaBeans
 * Activation Framework. This provides a more robust comparison
 * than is normally available in the DataFlavor class.
 *
 * @param mimeType        the MIME type
 * @return                true if the same MIME type
 */
 public boolean isMimeTypeEqual(String mimeType);

/**
 * Called on DataFlavor for every MIME Type parameter to allow
 * DataFlavor
 * subclasses to handle special parameters like the text/plain
 * charset parameters, whose values are case insensitive.  (MIME
 * type parameter values are supposed to be case sensitive).
 * <p>
 * This method is called for each parameter name/value pair and
 * should return the normalized representation of the
 * parameterValue.
 *
 * @param parameterName   the parameter name
 * @param parameterValue  the parameter value
 * @return                the normalized parameter value
 */
 protected String normalizeMimeTypeParameter(
                                      String parameterName,
                                      String parameterValue);

/**
 * Called for each MIME type string to give DataFlavor subtypes
 * the opportunity to change how the normalization of MIME types
 * is accomplished.
 * One possible use would be to add default parameter/value
 * pairs in cases where none are present in the MIME type string
 * passed in.
 *
 * @param mimeType        the MIME type
 * @return                the normalized MIME type
 */
 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

Mar. 6, 199: Version 1.0a

- Fixed minor typos.
- Synchronized with updated javadocs

12.0 Contacting Us

Please send your questions and comments to:

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