# JavaBeans Activation Framework Specification
*Draft 5*

**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 Java Standard Extension.**

## 1.0 Overview

JavaBeans is proving to be a popular technology. As more people embrace JavaBeans and Java 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.

Missing from JavaBeans and Java as a whole is 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 and some of the related architecture and implementation that supports these features.

Presumably with these pieces in place, a JavaBeans based component could be developed that provides helper application like functionality in a web browser, added functionality to an office suite, or a content viewer in a Java based network computer environment.

## 2.0 Goals

This document has the goal of describing a proposal for a Java implementation of a framework to provide the following services:

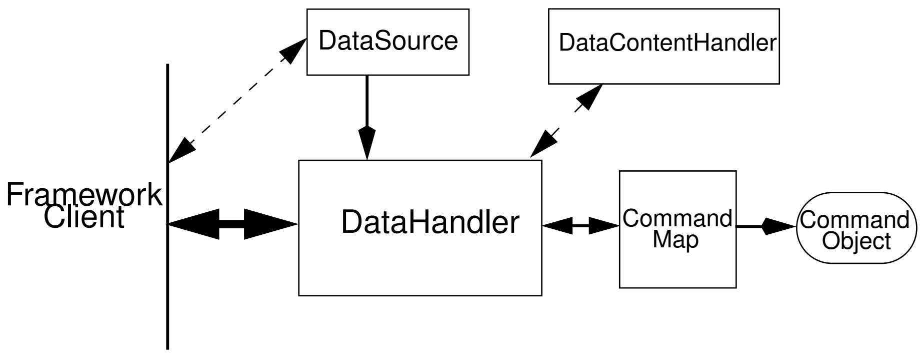
- A service to determine the type of arbitrary data.
- A service to encapsulate access to data.
- A service to discover the operations available on a particular type of data.

- A service to instantiate the correct software component that corresponds to the desired operation on a particular piece of data

This functionality will be packaged as a JDK 1.1.x and later compatible standard extension.

## 3.0 Architectural Overview

There already exists a fair amount of support in JDK 1.1 (including JavaBeans) to support a modest activation framework and the intention is to leverage as much of that existing technology as possible. The integration of these disparate pieces is the focus of this effort.




A diagram of the overall architecture is above. Note that the framework in the diagram is not bound to a particular application.

### 3.1 DataHandler

The object in the center of diagram is a class called the DataHandler. The data handler provides a consistent interface for the other subsystems to clients that wish to interface with this framework. The top of the diagram shows two ways arbitrary data could be introduced into the system, as a file and as a collection of bits.

### 3.2 DataSource

Data is encapsulated in an object implementing the DataSource interface which presents both a stream providing access to the data, and a String representing the MIME type of the data. Classes could be implemented for common data sources (web, file system, IMAP, ftp etc.). The DataSource interface could also be extended to allow per data source user customizations. Once the DataSource is set in the DataHandler, the operations available on that data can be determined.

### 3.3 CommandMap

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 would be able to 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 programming model for the software components that implemented the commands would be JavaBeans. These beans could use serialization, externalization or implement the ‘CommandObject’ interface to allow the typed data to be passed to them.

Our goal is to provide 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 might access the ‘types registry’ on the platform or use a server based solution. We will provide a simple default solution based on RFC 1524 (.mailcap) like functionality (see “Planned Deliverables” below).

### 3.4 Command Object

CommandObjects are JavaBeans that implement the CommandObject interface. The CommandObject interface allows beans that were implemented to be used in the framework to access their DataSource and DataHandler objects directly.

## 4.0 Using The Framework

The intent is to make this infrastructure widely available for any Java ‘application’ that needs this functionality. The ‘canonical’ consumer of this framework will access it through the DataHandler (although the major subsystems are designed to also operate independently). The underlying data source will be associated with the DataHandler when the DataHandler class is constructed. The DataHandler will get the data typing information from the DataSource or set directly from the constructor. Once this initialization step is complete, the consumer can request a list of commands that can be performed on that data item. When a request for this list is made the DataHandler uses the MIME type information of the data item to request a list of available commands from the CommandMap. The CommandMap has knowledge of available commands (implemented as JavaBeans) and their supported data types. The CommandMap will return to the DataHandler a subset of the full list of all commands based on the requested MIME type and the semantics of the CommandMap implementation. Ultimately when the application wishes to apply a command to some data it is done 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 Bean.

# 5.0 Usage Scenarios

This scenario is meant to provide the reader with a concrete example of how this framework might be used. The example application is a file system viewer in the spirit of CDE's 'dtfile' or similar to the Windows 95 Explorer. The basic functionality of the application is to present the user with a display of the available files. This hypothetical application has a function like CDE's dtfile or Explorer's 'right mouse' menu where all operations that can be performed on a data item are exposed in a popup menu for that item.

A typical user would use the application to view a directory of files. Upon finding a file of interest, the user could click on it which would bring up a popup menu which would list the available operations on that file. Operations commonly implemented would include 'edit', 'view' and 'print'. Selecting the 'view' operation for instance would cause the document to be opened in the appropriate viewer.

## 5.1 Scenario Architecture

The description of the application will be broken down into three discrete steps in the interest of clarity:

- Initialization: Application constructs a view of the file system.
- Get Command List: Application presents command list for a data item.
- Perform Command: Application performs command on a 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 would query the file system for its contents. For each file in the directory, a DataSource object would be instantiated. For each DataSource object a DataHandler is instantiated 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 would maintain a list of the DataHandler objects and query them for their names and icons to generate the 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 the files have been presented to the user, the user can click on a file which will bring up a popup menu that displays 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 uses the MIME Type of the data (which it gets from the DataSource object) to query the CommandMap for operations that are available on that type. The list can be interpreted and presented to the user in the form of a popup menu. The user is now free to select one of the operations from that list.

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

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, 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 method (DataHandler, Externalization etc.). Some CommandObjects (viewers for instance) will be subclassed from java.awt.Component and will require that they are given a parent container. Others (like a default print Command) may not present a UI. 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 Java-Beans 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 we view as the 'canonical' case. There are also circumstances where the application has already created Java Objects to represent its data. In this case creating an in memory instance of a DataSource that converted an existing object into an InputStream would at best be an inefficient use of system resources and could potentially result in a loss of data fidelity. In these cases an instance of DataHandler could be instantiated with the DataHandler(Object obj, String mimeType) constructor. DataHandler implements Transferable so the consuming beans can request representations other than InputStreams. The DataHandler will have to construct a DataSource for consumers that request it. The DataContentHandler mechanism will be extended to also allow conversion from Objects to InputStreams. The following is another example of a possible use of the framework. In this scenario a data base front end provides query results in terms of Java 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").getCommandOb-
ject());
  }
```

## 6.0 Primary Framework Interfaces

Based on the description of the overall architecture of this framework along with the diagram of that framework it is apparent that a number of interfaces need to be defined. This section will describe these interfaces.

### 6.1 DataSource

The data source 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. DataSource objects will likely be provided for common sources (file systems and URL's for instance) and application and system vendors will likely want to implement their own DataSources for things like IMAP servers, object databases, etc. 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. In the case of a file system a DataSource might use a simple mechanism such as file extensions to type data while a DataSource that supports incoming web based data may 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 DataHandler

The DataHandler is a class is used by clients of the framework to encapsulate the data source object and command object binding infrastructure. It encapsulates the type to command object binding service of the command map for applications. It provides a handle to the operations and data available on a data element. It also implements the Transferable interface. This allows applications and command objects to retrieve alternative representations (in the form of Java objects) of the underlying data. 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), an object (with associated content type) or a URL object. Once instantiated, the DataHandler will try to provide the 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 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 is explicitly set for the DataHandler. As a convenience, DataHandler uses the content type of it's 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 Java 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 it's 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 DataContentHandler

The DataContentHandler interface is used to write java objects used by the DataHandler to convert InputStreams into Java objects. Effectively the DataHandler uses this mechanism to implement the Transferable interface. DataContentHandlers will be specified in data files to refer to particular MIME types. DataFlavors are used to represent the types accessible from a DataContentHandler. We also provide an interface that allows us to write DataContentHandlers that go in the opposite direction. For instance an application may have an Image Object they wish to access as a gif file. The image.gif DataContentHandler would be used to convert the Image object into a gif format 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 CommandMap

Once the DataHandler has a MIME Type for the content, it can query the CommandMap for the operations, or *commands* that are available to that type. The application requests commands through the DataHandler (which in turn uses the CommandMap) to retrieve the JavaBean associated with that command. Some or all of the command map is stored in some ‘common’ place. One possible lightweight implementation might be a file like .mailcap (RFC 1524) file. One could imagine more featureful implementations that were perhaps distributed, or provided 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 BeanInfo 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 CommandInfo

The CommandInfo class is returned by the CommandMap to represent commands in the underlying registry. From the CommandInfo an application can instantiate the bean or request the verb 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 CommandObject

We expect JavaBeans that are designed specifically for use with the JavaBeans Activation Framework will implement the CommandObject interface. Specifically it gives beans direct access to methods in the DataHandler as well as notifying the bean which verb was used to call it.

```
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 DataContentHandlerFactory

Like the ContentHandler factory in the `java.net` package, the DataContentHandler-Factory is an interface that allows developers to write objects that map MIME types to DataContentHandlers. The interface is extremely simple, developers are free to implement what ever implementation they want.

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

## 7.0 Writing Beans for the Framework

### 7.1 Overview

This document describes the specification of well behaved viewers in the JavaBeans Activation Framework. It is important to note that this proposal is based heavily on JavaBeans and developers intending to implement viewers for the framework should be familiar with their basic concepts.

### 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. As a matter of fact, in many cases JavaBeans which weren't developed with knowledge of the framework can be used. Where possible we will exploit the existing features of JavaBeans and the JDK and define 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 might allow auto discovery of configuration files on the system. This would allow developers to deliver supplementary registry files to be appended to the system ones allowing additional packages to be added at runtime.

Our viewers/editors and related classes and files will be encapsulated into JAR files (as is the preferred method for beans). No restrictions will be made on which classes are used to implement beans beyond those expected of ‘well behaved’ beans.

## 7.4 Interfaces

Components can to implement the ‘CommandObject’ interface if they wish to communicate directly with their DataHandler and DataSource. The interface is small and easy to implement. Beans however can still use the traditional Serialization and Externalization methods available in JDK 1.1 and later.

## 7.5 Storage

As mentioned earlier, the canonical method of storage is via the DataHandler and DataSource. It is however possible to use Serialization and Externalization. An application that uses the framework could for instance implement the following:

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

The use of Serialized Objects is still being developed.

## 7.6 Packaging

The basic format for packaging of the Viewer/Editors is the JAR file as described in the JavaBeans specifications. This format allows the convenient packaging of collections of files that are related to a particular JavaBean or applet (see section 8 about Integration Points below).

## 7.7 Container Support

The JavaBeans Activation Framework has been designed to be flexible enough to support the needs of a variety of applications. It is expected that these applications will 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 we expect that the life cycle semantics of beans used in the framework will be the same as those for all JavaBeans. 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 implementation of the MailcapCommandMap will provide a mechanism to allow for an extensible set of command verbs. Applications using the framework will be able query the system for commands available for a particular MIME type and retrieve the bean associated with them.

## 8.0 Framework Integration Points

In an attempt to clarify how Beans developers can integrate with the JavaBeans Activation Framework, we present a number of scenarios.

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

- a mechanism for getting to the storage of data, 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, Beans

As a Bean developer, you're unlikely to need to develop a new DataSource or CommandMap. You might develop a DataContentHandler and of course you'll be building the visual Beans.

### 8.1 A 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 the Wombat file format in a language-independent manner, and 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 might do something like:

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

You'll 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 will be 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 might implement the Externalizable interface and use its methods to read and write your Wombat files. The DataHandler will arrange to 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 more heavy weight than the WombatViewer, but also allows editing.
- a WombatPrinter Bean that simply allows you to print a Wombat.
- a component that is responsible for reading and writing Wombat files.
- a Wombat class that encapsulates the Wombat data and is used by your other Beans and components.

In addition, there has been demand by customers to be able to programmatically manipulate Wombats, without necessarily using 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 will change to list the appropriate Wombat Beans as the implementations for the 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-ContentHandler=com.foo.WombatDataContentHandler
```

Your Wombat Beans could continue to implement the Externalizable interface, and thus read and write Wombat byte streams, but more likely they will simply operate on Wombat objects directly. To find the Wombat object they're being invoked to operate on, they will implement the CommandObject interface; the setCommandContext method will refer them to the corresponding DataHandler, from which they can invoke the getContent method, which will return a Wombat object (produced by the WombatDataContentHandler).

As before, 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 Content Handler 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've 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 will detect the presence of new Wombats, construct a WombatNotification object, and construct and send an email message with the Wombat notification data as an attachment. Your customers will run a program that scans their email INBOX for messages with Wombat notification attachments and use the WombatNotification class to notify the user 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 will create DataHandlers for its WombatNotification objects. The DataHandler will be used by the email system to fetch a byte stream corresponding to the WombatNotification object. (The DataHandler will use the DataContentHandler to do this.)

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

# 9.0 Framework Deliverables

## 9.1 Packaging Details

The desire to have this functionality available on JDK 1.1 has caused us to implement it as a Standard Extension. This will allow it 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 will be `javax.activation.`
- The initial release will be supported on JDK 1.1.x (exact revision TBD) 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 Transferable. Additionally it provides access to the registry infrastructure that 'discovers' available beans.

**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 the registry.

**interface CommandObject:** An interface that can be implemented by beans 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 will use 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 beans. (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 MimeType:** This class provides RFC 2046 MIME Type parsing.

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

**class com.sun.activation.DataContentHandlers.*:** some default data content handlers.

# 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: Forth Public Draft 5

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

- **Note:** Change bars reflect differences from the October 28 draft.

## 12.0 Contacting Us

Please send your questions and comments to:

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