JavaBeans™ Activation Framework Specification *Version 1.1*

Bart Calder, Bill Shannon

The JavaBeans™ Activation Framework Specification is a data typing and registry technology that is a Standard Extension to the Java™ Platform.

Specification: JavaBeans Activation Framework version 1.1 Specification ("Specification")

Status: Maintenance Release

Release: 17 April 2006

LIMITED LICENSE GRANTS

1. License for Evaluation Purposes. Sun hereby grants you a fully-paid, non-exclusive, non-transferable, worldwide, limited license (without the right to sublicense), under Sun's applicable intellectual property rights to view, download, use and reproduce the Specification only for the purpose of internal evaluation. This includes (i) developing applications intended to run on an implementation of the Specification, provided that such applications do not themselves implement any portion(s) of the Specification, and (ii) discussing the Specification with any third party; and (iii) excerpting brief portions of the Specification in oral or written communications which discuss the Specification provided that such excerpts do not in the aggregate constitute a significant portion of the Specification.

2. License for the Distribution of Compliant Implementations. Sun also grants you a perpetual, non-exclusive, non-transferable, worldwide, fully paid-up, royalty free, limited license (without the right to sublicense) under any applicable copyrights or, subject to the provisions of subsection 4 below, patent rights it may have covering the Specification to create and/or distribute an Independent Implementation of the Specification that: (a) fully implements the Specification including all its required interfaces and functionality; (b) does not modify, subset, superset or otherwise extend the Licensor Name Space, or include any public or protected packages, classes, Java interfaces, fields or methods within the Licensor Name Space other than those required/authorized by the Specification or Specifications being implemented; and (c) passes the Technology Compatibility Kit (including satisfying the requirements of the applicable TCK Users Guide) for such Specification ("Compliant Implementation"). In addition, the foregoing license is expressly conditioned on your not acting outside its scope. No license is granted hereunder for any other purpose (including, for example, modifying the Specification, other than to the extent of your fair use rights, or distributing the Specification to third parties). Also, no right, title, or interest in or to any trademarks, service marks, or trade names of Sun or Sun's licensors is granted hereunder. Java, and Java-related logos, marks and names are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries.

3. Pass-through Conditions. You need not include limitations (a)-(c) from the previous paragraph or any other particular “pass through” requirements in any license You grant concerning the use of your Independent Implementation or products derived from it. However, except with respect to Independent Implementations (and products derived from them) that satisfy limitations (a)-(c) from the previous paragraph, You may neither: (a) grant or otherwise pass through to your licensees any licenses under Sun’s applicable intellectual property rights; nor (b) authorize your licensees to make any claims concerning their implementation’s compliance with the Specification in question.

4. Reciprocity Concerning Patent Licenses.

a. With respect to any patent claims covered by the license granted under subparagraph 2 above that would be infringed by all technically feasible implementations of the Specification, such license is conditioned upon your offering on fair, reasonable and non-discriminatory terms, to any party seeking it from You, a perpetual, non-exclusive, non-transferable, worldwide license under Your patent rights which are or would be infringed by all technically feasible implementations of the Specification to develop, distribute and use a Compliant Implementation.

b With respect to any patent claims owned by Sun and covered by the license granted under subparagraph 2, whether or not their infringement can be avoided in a technically feasible manner when implementing the Specification, such license shall terminate with respect to such claims if You initiate a claim against Sun that it has, in the course of performing its responsibilities as the Specification Lead, induced any other entity to infringe Your patent rights.

c Also with respect to any patent claims owned by Sun and covered by the license granted under subparagraph 2 above, where the infringement of such claims can be avoided in a technically feasible manner when implementing the Specification such license, with respect to such claims, shall terminate if You initiate a claim against Sun that its making, having made, using, offering to sell, selling or importing a Compliant Implementation infringes Your patent rights.

5. Definitions. For the purposes of this Agreement: “Independent Implementation” shall mean an implementation of the Specification that neither derives from any of Sun’s source code or binary code materials nor, except with an appropriate and separate license from Sun, includes any of Sun’s source code or binary code materials; “Licensor Name Space” shall mean the public class or interface declarations whose names begin with “java”, “javax”, “com.sun” or their equivalents in any subsequent naming convention adopted by Sun through the Java Community Process, or any recognized successors or replacements thereof; and “Technology Compatibility Kit” or “TCK” shall mean the test suite and accompanying TCK User’s Guide provided by Sun which corresponds to the Specification and that was available either (i) from Sun’s 120 days before the first release of Your Independent Implementation that allows its use for commercial purposes, or (ii) more recently than 120 days from such release but against which You elect to test Your implementation of the Specification.

This Agreement will terminate immediately without notice from Sun if you breach the Agreement or act outside the scope of the licenses granted above.

DISCLAIMER OF WARRANTIES

THE SPECIFICATION IS PROVIDED “AS IS”. SUN MAKES NO REPRESENTATIONS OR WARRANTIES, EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO, WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, NON-INFRINGEMENT (INCLUDING AS A CONSEQUENCE OF ANY PRACTICE OR IMPLEMENTATION OF THE SPECIFICATION), OR THAT THE CONTENTS OF THE SPECIFICATION ARE SUITABLE FOR ANY PURPOSE. This document does not represent any commitment to release or implement any portion of the Specification in any product. In addition, the Specification could include technical inaccuracies or typographical errors.

LIMITATION OF LIABILITY

TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL SUN OR ITS LICENSORS BE LIABLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION, LOST REVENUE, PROFITS OR DATA, OR FOR SPECIAL, INDIRECT, CONSEQUENTIAL, INCIDENTAL OR PUNITIVE DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF OR RELATED IN ANY WAY TO YOUR HAVING, IMPLEMENTING OR OTHERWISE USING THE SPECIFICATION, EVEN IF SUN AND/OR ITS LICENSORS HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

You will indemnify, hold harmless, and defend Sun and its licensors from any claims arising or resulting from: (i) your use of the Specification; (ii) the use or distribution of your Java application, applet and/or implementation; and/or (iii) any claims that later versions or releases of any Specification furnished to you are incompatible with the Specification provided to you under this license.

RESTRICTED RIGHTS LEGEND

U.S. Government: If this Specification is being acquired by or on behalf of the U.S. Government or by a U.S. Government prime contractor or subcontractor (at any tier), then the Government’s rights in the Software and accompanying documentation shall be only as set forth in this license; this is in accordance with 48 C.F.R. 227.7201 through 227.7202-4 (for Department of Defense (DoD) acquisitions) and with 48 C.F.R. 2.101 and 12.212 (for non-DoD acquisitions).

REPORT

If you provide Sun with any comments or suggestions concerning the Specification (“Feedback”), you hereby: (i) agree that such Feedback is provided on a non-proprietary and non-confidential basis, and (ii) grant Sun a perpetual, non-exclusive, worldwide, fully paid-up, irrevocable license, with the right to sublicense through multiple levels of sublicensees, to incorporate, disclose, and use without limitation the Feedback for any purpose.

GENERAL TERMS

Any action related to this Agreement will be governed by California law and controlling U.S. federal law. The U.N. Convention for the International Sale of Goods and the choice of law rules of any jurisdiction will not apply.

The Specification is subject to U.S. export control laws and may be subject to export or import regulations in other countries. Licensee agrees to comply strictly with all such laws and regulations and acknowledges that it has the responsibility to obtain such licenses to export, re-export or import as may be required after delivery to Licensee.

This Agreement is the parties' entire agreement relating to its subject matter. It supersedes all prior or contemporaneous oral or written communications, proposals, conditions, representations and warranties and prevails over any conflicting or additional terms of any quote, order, acknowledgment, or other communication between the parties relating to its subject matter during the term of this Agreement. No modification to this Agreement will be binding, unless in writing and signed by an authorized representative of each party.

Rev. April, 2006

Sun/Final/Full

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™ application.

2.0 Goals

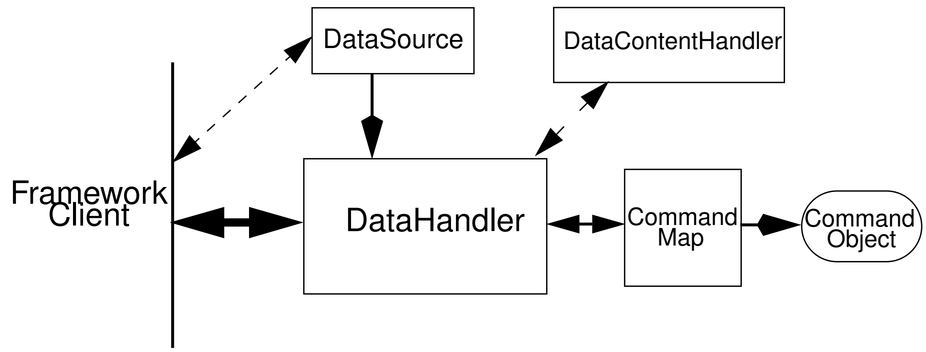
This document describes the JavaBeans™ Activation Framework (JAF). The JAF implements the following services:

- 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

The Java™ platform (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 implement 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, 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 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.

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 that 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 the Windows Explorer utility. When launched, it presents the user with a display of available files. It includes a function like 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 that 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 that 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. 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 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.

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) or as an object with an associated content type.

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.

6.3 The DataContentHandler Interface

The DataContentHandler interface is implemented by classes that are used by the DataHandler to convert InputStreams into objects and vice versa. 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 image/gif DataContentHandler converts the image object into a gif-formatted byte stream.

Applications will typically need to provide DataContentHandlers for all the MIME types they intend to support. (Note that the JavaMail reference implementation provides DataContentHandlers for many of the MIME types used in mail messages.)

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.

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.

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.

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.

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. That is, require 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 beyond ‘generic’ Beans. In many cases, JavaBeans™ components that 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 viewers/editors will be bound to data via a simple registry mechanism similar in function to a .mailcap file. In addition, mailcap format files may be bundled with components, 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.

7.4 Interfaces

A viewer Bean that communicates directly with a JAF DataHandler 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(
                                data_handler.getOutputStream());
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 called 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 that 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/WombatEditBean.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 your 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 and is also included in the J2SE 6.0 release. The following are some more details about the package:

- The package name is `javax.activation.`
- This release is included in J2SE 6.0 and J2EE 5.0, and is also available separately for use on earlier versions of J2SE.
- The JAF Reference Implementation does not include DataContentHandlers for any MIME data types; applications must include the DataContentHandlers thy need.

Note that the JavaMail Reference Implementation includes DataContentHandlers for some basic data types used in mail messages.

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 implementations 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.

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.

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 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, 1999: Version 1.0a

- Fixed minor typos.
- Synchronized with updated javadocs

May 16, 2005

- Updated for 1.1 release
- Removed all detailed API specifications; refer to the javadocs

11.0 Contacting Us

Please send your questions and comments to:

`activation-comments@sun.com`