JavaSoft

JavaBeans™

This is the JavaBeans™ 1.0 API specification.

It describes the core specification for the JavaBeans component architecture. See the change history on page 105 for some minor changes since the October version of the spec.

The 1.0 specification is now frozen, but we are still interested in feedback on the JavaBeans architecture for future releases. Because of the volume of interest in JavaBeans we can't respond individually to reviewer comments, but we do carefully read and consider all reviewer input. Please send comments to java-beans@java.sun.com.

To stay in touch with the JavaBeans project, visit our web page at:

http://java.sun.com/beans

Java Beans 1.0

Version 1.00-A

December 4, 1996

Contents

Acknowledgments

The JavaBeans APIs are a broad effort that include contributions from numerous groups at Sun and at partner companies. Many of the APIs referenced here have broader applicability than simply JavaBeans, and we'd like to thank their authors for helping to align them with the JavaBeans framework.

We would like to thank everyone who has contributed, but especially those authors who have drafted or proposed major chunks of the APIs described here, including Tom Ball, Jeff Bonar, Larry Cable, Amy Fowler, Graham Hamilton, Noah Mendelsohn, Eduardo Pelegri-Llopart, Carl Quinn, Jim Rhyne, Roger Riggs, John Rose, Nakul Saraiya, and Blake Sullivan.

We'd also like to thank the many reviewers who have provided comments and suggestions on the various drafts of the specs. The public review process was extremely valuable and the many thoughtful comments we received helped to substantially improve the Java Beans design.

Related Documents

This document describes the main APIs for JavaBeans. However the JavaBeans APIs are also closely related to other new Java core APIs which are described in separate documents.

For example, JavaBeans uses the new Java Object Serialization API as its main persistence mechanism, and that mechanism has been designed to work well with beans. Similarly, JavaBeans makes heavy use of the new Java Core Reflection API, etc.

For the current status and pointers to on-line copies, check the "Related APIs" link on our JavaBeans home page or go to:

```
http://java.sun.com/beans/related.html
```

- "*The Java Core Reflection API*". This describes the low-level reflection API that is used to provide the Introspection information described in Section 8.
- "*Java Object Serialization*". This describes the automatic Java Object Serialization mechanism that is the default mechanism for Java Beans persistence (see Section 5).
- "*JDK 1.1 - AWT Enhancements*". Changes to AWT, including support for drag-and-drop and cut-and-paste (see Section 4).
- "*JAR file specification*". This is the standard packaging mechanism for Java Beans. (See Section 11.)
- "*Remote Method Invocation*". This specifies the Java RMI distributed programming mechanism.
- *"Java IDL"*. This describes the Java mapping to the industry standard CORBA IDL system.

1 Introduction

The goal of the JavaBeans APIs is to define a *software component model* for Java, so that third party ISVs can create and ship Java components that can be composed together into applications by end users.

1.1 Component granularity

There are a range of different kinds of JavaBeans components:

1. Some JavaBeans will be used as building blocks in composing applications. So a user may be using some kind of builder tool to connect together and customize a set of Java Beans to act as an application. Thus for example, an AWT button would be a Java Bean.
2. Some JavaBeans will be more like regular applications, which may then be composed together into compound documents. So a Java Bean spreadsheet might be embedded inside a Web page.

These two aspects overlap. For example a spreadsheet might live within a composite application as well as within a more normal compound document. So there is more of a continuum than a sharp cutoff between "composite applications" and "compound documents".

The design centre for beans ranges from small controls up through simple compound documents such as Web pages. However we are not currently trying to provide the kind of high-end document integration that is typical of full function document systems such as ClarisWorks or Microsoft Office. Thus we provide APIs that are analogous to (say) the OLE Control or ActiveX APIs, but we to do not try to provide the full range of high-end document APIs provided by (for example) OpenDoc. However, we do intend to allow beans to be embedded as components in high-end compound documents, and we also expect that some key document suite components (such as word processors and spreadsheets) will also be supported as Java Beans.

In general we expect that most Java Bean components will be small to medium sized controls and that we should make the simple cases easy and provide reasonable defaults for as much behaviour as possible.

1.2 Portability

One of the main goals of the Java Beans architecture is to provide a platform neutral component architecture. When a Java Bean is nested inside another Java Bean then we will provide a full functionality implementation on all platforms. However, at the top level when the root Java Bean is embedded in some platform specific container (such as Word or Visual Basic or ClarisWorks or Netscape Navigator) then the Java Beans APIs should be integrated into the platform's local component architecture.

For example, this means that on the Microsoft platforms the Java Beans APIs will be bridged through into COM and ActiveX. Similarly, it will be possible to treat a bean as a Live Object (née OpenDoc) part, or to integrate a bean with LiveConnect inside Netscape Navigator.

So a single Java Bean should be capable of running in a wide range of different environments. Within each target environment it should be able to fire events, service method invocations, etc., just like any other component.

We do not specifically discuss the different bridges as part of this document. The existence of the bridges is transparent to Java Bean developers and it is the bridges' responsibility to take the Java Beans APIs defined in this document and bridge them through to the platform specific component architecture.

However the need to bridge to these other component models (notably OpenDoc, OLE/COM/ActiveX, and LiveConnect) has been one of the constraints in designing the Java Beans APIs. We have been careful to ensure that the various beans APIs can be cleanly converted to these three main component models.

1.3 A uniform, high-quality API

Different platforms will vary in their ability to support the full Java Bean APIs. However whenever a platform is unable to provide the full functionality it must provide some reasonable, harmless default instead.

So for example, if a platform doesn't support menubar-merging then when a Java Bean component provides a menubar it may pop up above the component, rather than being merged into the containing document's menubar.

This means that Java Bean component writers can program to a consistent set of APIs and trust them to work everywhere. We don't want bean implementors to have to do checks to discover which facilities are supported on their current platform.

1.4 Simplicity

We would like to keep the Java Bean APIs relatively simple. We will focus on making small lightweight components easy to implement and use, while making heavyweight components possible.

As a general rule we don't want to invent an enormous java.beans.Everything class that people have to inherit from. Instead we'd like the Java Bean runtimes to provide default behaviour for "normal" objects, but to allow objects to override a given piece of default behaviour by inheriting from some specific java.beans.something interface.

One of our goals is that people should be able to learn the basic Java Beans concepts very quickly so that they can start writing and using simple components with very little effort and then slowly progress to using the more sophisticated features of the API.

2 Fundamentals

2.1 What is a Bean?

Let's start with an initial definition and then refine it:

> **"A Java Bean is a reusable software component that can be manipulated visually in a builder tool."**

This covers a wide range of different possibilities.

The builder tools may include web page builders, visual application builders, GUI layout builders, or even server application builders. Sometimes the "builder tool" may simply be a document editor that is including some beans as part of a compound document.

Some Java Beans may be simple GUI elements such as buttons and sliders. Other Java Beans may be sophisticated visual software components such as database viewers, or data feeds. Some Java Beans may have no GUI appearance of their own, but may still be composed together visually using an application builder.

Some builder tools may operate entirely visually, allowing the direct plugging together of Java Beans. Other builders may enable users to conveniently write Java classes that interact with and control a set of beans. Other builders may provide a simple scripting language to allow easy high-level scripting of a set of beans.

Individual Java Beans will vary in the functionality they support, but the typical unifying features that distinguish a Java Bean are:

- Support for "introspection" so that a builder tool can analyze how a bean works
- Support for "customization" so that when using an application builder a user can customize the appearance and behaviour of a bean.
- Support for "events" as a simple communication metaphor than can be used to connect up beans.
- Support for "properties", both for customization and for programmatic use.
- Support for persistence, so that a bean can be customized in an application builder and then have its customized state saved away and reloaded later.

Note that while beans are primarily targeted at builder tools they are also entirely usable by human programmers. All the key APIs such as events, properties, and persistence, have been designed to work well both for human programmers and for builder tools.

Many beans will have a strong visual aspect, in both the application builder and in the final constructed application, but while this is common it is not required.

2.1.1 Beans v. Class Libraries

Not all useful software modules should necessarily turn into beans. Beans are appropriate for software components that can be visually manipulated and customized to achieve some effect. Class libraries are an appropriate way of providing functionality that is useful to programmers, but which doesn't benefit from visual manipulation.

So for example it makes sense to provide the JDBC database access API as a class library rather than as a bean, because JDBC is essentially a programmatic API and not something that can be directly presented for visual manipulation. However it does make sense to write database access beans on top of JDBC. So for example you might write a "select" bean that at customization time helped a user to compose a select statement, and then when the application is run uses JDBC to run the select statement and display the results.

2.2 Properties, events, and methods

The three most important features of a Java Bean are the set of *properties* it exposes, the set of *methods* it allows other components to call, and the set of *events* it fires.

Properties are described in more detail in Section 7. Basically properties are named attributes associated with a bean that can be read or written by calling appropriate methods on the bean. Thus for example, a bean might have a "foreground" property that represents its foreground color. This property might be read by calling a "Color getForeground()" method and updated by calling a "void setForeground(Color c)" method.

The *methods* a Java Bean exports are just normal Java methods which can be called from other components or from a scripting environment. By default all of a bean's public methods will be exported, but a bean can choose to export only a subset of its public methods (see Section 8.5).

Events are described in more detail in Section 6. Events provide a way for one component to notify other components that something interesting has happened. Under the new AWT event model an event listener object can be registered with an event source. When the event source detects that something interesting happens it will call an appropriate method on the event listener object.

2.3 Design time v. run-time

Each Java Bean component has to be capable of running in a range of different environments. There are really a continuum of different possibilities, but two points are particularly worth noting.

First a bean must be capable of running inside a builder tool. This is often referred to as the *design environment*. Within this design environment it is very important that the bean should provide design information to the application builder and allow the end-user to *customize* the appearance and behaviour of the bean.

Second, each bean must be usable at run-time within the generated application. In this environment there is much less need for design information or customization.

The design time information and the design time customization code for a component may potentially be quite large. For example, if a component writer provides a "wizard" style customizer that guides a user through a series of choices, then the customization code may easily dwarf the run-time code for the bean. We therefore wanted to make sure that we have a clear split between the design-time aspects of a bean and the run-time aspects, so that it should be possible to deploy a bean at run-time without needing to download all its design time code.

So, for example, we allow the design time interfaces (described in chapters 8 and 9) to be supported in a separate class from the run-time interfaces (described in the other chapters).

2.4 Security Issues

Java Beans are subject to the standard Java security model. We have neither extended nor relaxed the standard Java security model for Java Beans.

Specifically, when a Java Bean runs as part of an untrusted applet then it will be subject to the standard applet security restrictions and won't be allowed to read or write arbitrary files, or to connect to arbitrary network hosts. However when a Java Bean runs as part of a stand-alone Java application, or as part of a trusted (signed) applet, then it will be treated as a normal Java application and allowed normal access to files and network hosts.

In general we advise Java Bean developers to design their beans so that they can be run as part of untrusted applets. The main areas where this shows up in the beans APIs are:

- *Introspection.* Bean developers should assume that they have unlimited access to the high level Introspection APIs (Section 8) and the low-level reflection APIs in the design-time environment, but more limited access in the run-time environment. For example, the standard JDK security manager will allow trusted applications access to even private field and methods, but will allow untrusted applets access to only public fields and methods. (This shouldn't be too constraining - the high-level Introspection APIs only expose "public" information anyway.)
- *Persistence.* Beans should expect to be serialized or deserialized (See Section 5) in both the design-time and the run-time environments. However in the run-time environment, the bean should expect the serialization stream to be created and controlled by their parent application and should not assume that they can control where serialized data is read from or written to. Thus a browser might use serialization to read in the initial state for an untrusted applet, but the applet should not assume that it can access random files.
- *GUI Merging.* In general untrusted applets will not be permitted to perform any kind of GUI merging with their parent application. So for example, menubar merging might occur between nested beans inside an untrusted applet, but the top level menubar for the untrusted applet will be kept separate from the browser's menubar.

None of these restrictions apply to beans running as parts of full-fledged Java applications, where the beans will have full unrestricted access to the entire Java platform API.

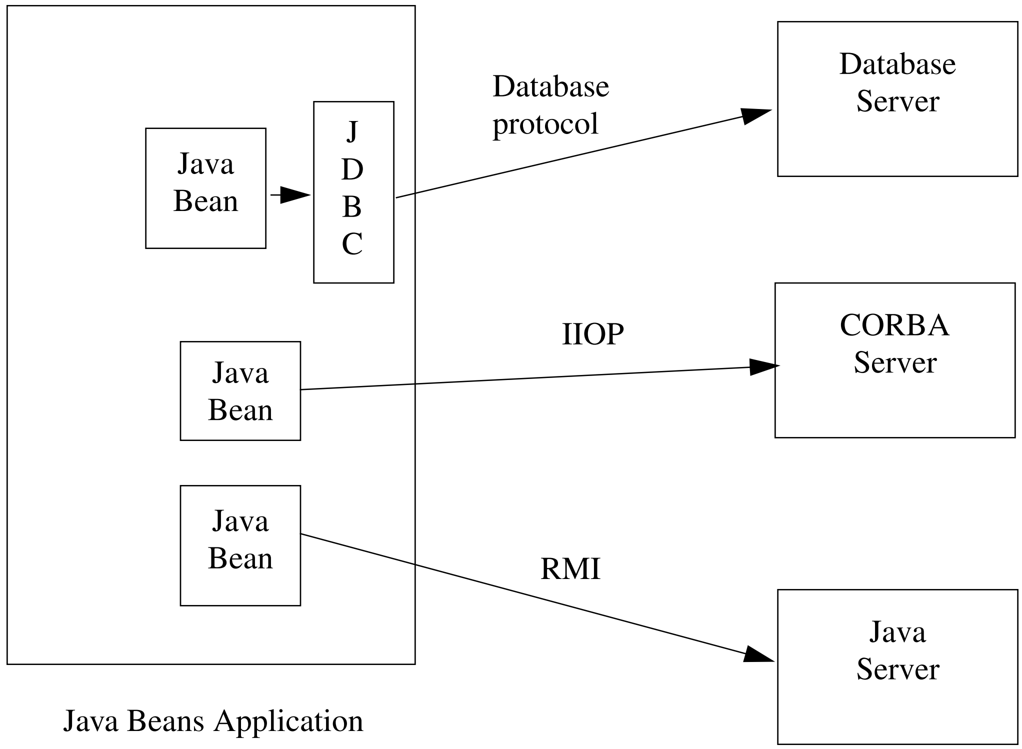
2.5 Local activation

The basic run-time model for Java Bean components is that they run within the same address space as their container.

So for example, if the container is a Java application, then the contained bean is run in the same Java virtual machine as its container. If the container is a non-Java application, then the Java Bean will run in a Java virtual machine that is directly associated with the application. (Normally this virtual machine will be running in the same address space as the application.)

2.6 Java Beans as links to remote state

The Java Beans architecture is being designed to work well in a distributed world-wide-web environment.



A key part of designing distributed systems is engineering a good split between local and remote processing. Local processing can benefit from rapid communication with a single machine, while remote accesses may have long latencies and suffer a variety of communication failures. Designers of distributed systems tend to design their remote interfaces very carefully, to minimize the number of remote interactions and to use various kinds of data caching and update batching to reduce remote traffic.

Therefore, we recommend that distributed system implementors design beans to act as intelligent front-ends to talk back to their network servers. So rather than designing all beans APIs to work over the network, we are primarily targeting the beans APIs for use within a virtual machine, where interactions are cheap, and we provide several alternative mechanisms to allow beans developers to connect back to network servers.

The three primary network access mechanisms that will be available to Java Beans developers on all Java platforms are:

- *Java RMI*. The Java Remote Method Invocation facility makes it very easy to develop distributed Java applications. The distributed system interfaces can be designed in Java and clients and servers can be implemented against those interfaces. Java RMI calls will be automatically and transparently delivered from the client to the server. We expect that the use of Java RMI will be a very common way for writing Java Beans within networked systems.

- *Java IDL*. The Java IDL system implements the industry standard OMG CORBA distributed object model. All the system interfaces are defined in the CORBA IDL interface definition language. Java stubs can be generated from these IDL interfaces, allowing Java Bean clients to call into IDL servers, and vice versa. CORBA IDL provides a multi-language, multi-vendor distributed computing environment, and the use of Java IDL allows Java Bean clients to talk to both Java IDL servers and other non-Java IDL servers.
- *JDBC*. The Java database API, JDBC, allows Java Bean components access to SQL databases. These database can either be on the same machine as the client, or on remote database servers. Individual Java Beans can be written to provide tailored access to particular database tables.

Another solution to distributed computing is to migrate objects around the network. Thus an expense voucher bean might be created on one user's workstation and then get forwarded around the network to various interim severs and to various other workstations for review, and then get forwarded to an accounts payable server to be processed for final payment. In this scenario, the bean will show a GUI appearance at some points in its life, but also be run invisibly within server applications at other times.

2.7 Invisible beans

Many Java Beans will have a GUI representation. When composing beans with a GUI application builder it may often be this GUI representation that is the most obvious and compelling part of the beans architecture.

However it is also possible to implement invisible beans that have no GUI representation. These beans are still able to call methods, fire events, save persistent state, etc. They will also be editable in a GUI builder using either standard property sheets or customizers (see Chapter 9). They simply happen to have no screen appearance of their own.

Such invisible beans can be used either as shared resources within GUI applications, or as components in building server applications that have no GUI appearance at all.

These invisible beans may still be represented visually in an application builder tool, and they may have a GUI customizer that configures the bean.

Some beans may be able to run either with or without a GUI appearance depending where they are instantiated. So if a given bean is run in a server it may be invisible, but if it is run on a user's desktop it may have a GUI appearance.

2.8 Multi-Threading

Java Beans should assume that they are running in a multi-threaded environment and that several different threads may be simultaneously delivering events and/or calling methods and/or setting properties.

It is the responsibility of each java bean developer to make sure that their bean behaves properly under multi-threaded access. For simple beans this can generally be handled by simply making all the methods "synchronized".

2.9 Internationalization

As part of the JDK 1.1, various internationalization APIs are being added to the core Java API set. These internalization APIs can be used by individual Java Beans.

In general the Java internationalization model is that individual packages are responsible for internally localizing any appropriate strings using the current default locale (from java.util.Locale.getDefault) and that appropriately localized strings are passed across public APIs. For example it would be the Color.toString() method's responsibility to generate a localized color name such as "rouge" in the French locale, rather than it being a client's responsibility to take a locale independent string "red" and then try to localize it.

One particular issue for Java Beans is the naming of events, methods, and properties. In general we expect Java programs to work programmatically in terms of locale-independent *programmatic* names.

The two main exceptions to this are the "display name" and "short help" strings supported by the FeatureDescriptor class. These should be set to localized strings.

So for example, a Java Bean might export a method whose locale-independent name is "hello" but whose display name is localized in the glaswegian locale as "heyJimmie". Builder tools may present this locale-dependent display name "heyJimmie" to human users and this name may be usable from scripting environments. However any generated Java code must use the locale-independent programmatic method name "hello".

Similarly, a property may have the locale-independent programmatic name "cursorColour" and be accessed by methods getCursorColour and setCursorColour, but may be localized in the U.S to have display name "cursorColor". When the property is presented to humans via property sheets, etc., it may be named as "cursorColor", but the getter/setter methods are still accessed via their local-independent names getCursorColour and setCursorColour.

3 Scenarios

To help illustrate how beans are intended to function this section walks through two sample scenarios of bean use.

Note that this section is intended to be illustrative rather than prescriptive.

It is intended to show some ways beans may be used, but not to describe all possibilities. In particular different application builder tools are likely to offer very different application building scenarios.

The two scenarios involve constructing applets, but the same steps apply to building full-scale java applications.

3.1 Using an application builder to construct an Applet

In this scenario a user uses an application builder to create an applet. The application builder generates various pieces of source code and provides the user with templates into which the user can add their own source code.

1. Buying the tools.

1.a) The user buys and installs the WombatBuilder Java application builder.

1.b) The user buys some Java Bean components from Al's Discount Component Store. They get a win32 floppy containing the beans in a JAR file. This includes a "Button" component and a "DatabaseViewer" component.

1.c) The user inserts the floppy into the machine and uses the WombatBuilder "Add bean..." dialog to tell WombatBuilder about the beans. WombatBuilder copies the beans off to its bean collection. (WombatBuilder may either introspect on the beans at this point to learn about their attributes or it may wait until the beans are actually used.)

2. Laying out the applet

2.a) The user starts to create a new custom applet called "Foobaz" in WombatBuilder. Initially they start with a blank applet template. WombatBuilder creates a Foobaz.java file to represent the state of this applet.

2.b) The user selects a "button" from the WombatBuilder beans menu and drops it onto the applet template. WombatBuilder uses "new Button()" to allocate a new Button instance. WombatBuilder then adds the new button as a member field to the Foobaz class.

2.c) The user selects a "database viewer" from the WombatBuilder beans menu and drops it onto the applet template. Because the database viewer is a fairly complex bean it comes with some serialized state to pre-initialize itself. The WombatBuilder tool therefore uses the Beans.instantiate utility method to read in a serialized object instance. WombatBuilder then adds this new database viewer object as a member field to the Foobaz class.

3. Customizing the beans.

3.a) The user selects the button and asks WombatBuilder to edit it. WombatBuilder uses Introspector.getBeanInfo to obtain a BeanInfo object describing the bean and discovers that this bean doesn't have a Customizer. So it uses the BeanInfo information to learn the names of the bean's read-write properties and generates a simple property sheet containing the names of the properties and their current values. The user then interacts with the property sheet to customize the button's appearance. The updates are applied back to the bean as they happen.

3.b) The user selects the database viewer and asks WombatBuilder to edit it. Using Introspector.getBeanInfo, WombatBuilder discovers that the database viewer comes with its own bean Customizer class. WombatBuilder instantiates an object of the customizer class and asks it to customize the component. The database viewer customizer brings up a series of dialogs that guide the user through the steps of selecting a target database and identifying tables and records to display. The database viewer customizer uses a set of package private methods to setup the state of the database viewer bean.

4. Connecting up the beans

4.a) The user now decides that they want to hook-up the button to prod the database viewer. They select the button on the screen and pull-down the WombatBuilder “events” menu to discover its events and then select the “action” event. WombatBuilder creates a template method “Foobaz.button1Action” and brings up a text editor so that the user can type in the text of the method.

4.b) The user types in the text of the method. In this case they simply call the database viewer component's “update” method. So now the automatically generated Foobaz looks like:

```
class Foobaz extends WombatBuilderApplet {
    public Button button1;
    public DatabaseViewer databaseViewer1;
    void button1Action(java.awt.event.ActionEvent evt) {
        databaseViewer1.update();
    }
}
```

4.c) In addition, behind the scenes, WombatBuilder creates an adaptor class to connect the button to the Foobaz. This looks something like:

```
class Hookup1234 implements java.awt.event.ActionListener {
    public Foobaz target;
    public void actionPerformed(java.awt.event.ActionEvent evt) {
        target.button1Action(evt);
    }
}
```

4.d) WombatBuilder creates a new instance of the Hookup1234 class and fills in its “target” field with the target applet. It then takes the hookup object and registers it with the button1 “addActionListener” method. This completes the hookup.

5. Packaging up the applet

5.a) The user decides they are done and pushes “make it so”. WombatBuilder uses Java serialization to store away the Foobaz object in a Foobaz.ser pickle file. As part of storing away the Foobaz object, the two beans instances (the button1 and databaseViewer1 members) also get stored away. The WombatBuilder also generates some initialization code to reconstruct the event hookup.

5.b) WombatBuilder creates a Foobaz.jar file holding the pickle and the .class files needed by the applet including the hookup1234 .class file and the .class files for the beans themselves.

5.c) WombatBuilder also generates a sample HTML page that uses the applet tag to load the pickled Foobaz.ser from the Foobaz.jar file.

6. Trying it out

6.a) The user then tests out the applet in their SpaceWars web browser. When they visit the html page, the applet is initialized by reading in its serialized state from the Foobaz.ser inside of the Foobaz.jar file. As part of doing this various classes are also loaded from the .class files inside the Foobaz.jar file.

6.b) The result is a running applet with a working event hookup from the button to the database viewer, and with its state setup in the way that the user customized it in the WombatBuilder tool.

6.c) Joy, Happiness, and Banana Mochas all round.

3.2 Using beans in a hand-written Applet

In this scenario we use some Java beans components in a hand-written Applet. This scenario is fairly similar to building an Applet with simple AWT components today, and is intended to illustrate how beans can be used without special tools.

1. Loading the tools.

1.a) The user downloads and installs the JDK, including the javac compiler.

1.b) The user notices some useful Java Bean components on Dave’s Freeware Components Page. They ftp over a beans jar file.

1.c) The user uses the “jar” command to extract the contents of the jar file and put them in the user’s normal “class” directory. This includes a “Button” component and a “DatabaseViewer” component.

2. Laying out the applet

2.a) The user starts to type in the code for a new applet. In the Applet’s “init” method they create a dave.Button and a dave.DatabaseViewer. For the dave.Button they can simply do a “new”. However, the dave.DatabaseViewer comes with some serialized state, so they must use Beans.instantiate to read in the serialized object.

```
class MyApplet extends java.awt.Applet {

    public void init() {
        btn = new dave.Button();
        add(btn);
        try {
            ClassLoader cl = Class.forName(
                         "dave.DatabaseViewer").getClassLoader();
            dbViewer = Beans.instantiate(cl,
                             "dave/DatabaseViewer.ser");
            add(dbViewer);
        } catch (Exception ex) {
            throw new Error("Couldn't make dbViewer");
        }
    }

    private DatabaseViewer dbViewer;
    private Button btn;
}
```

3. Customizing the beans.

3.a) The user decides they want to tweak the appearance and behaviour of the components a little, so in their MyApplet.init method they add some calls on the beans' property accessor methods:

```
btn.setBackground(java.awt.Color.red);
dbViewer.setURL("jdbc:msql://aardvark.eng/stuff");
```

4. Connecting up the beans

4.a) The user now decides that they want to hook-up the button to prod the database viewer. The button fires its events at the java.awt.event.ActionListener interface. Since they only have a simple applet with a single button, they decide to make their MyApplet class implement the ActionListener directly and they add an "actionPerformed" method as required by the ActionListener interface:

```
void actionPerformed(java.awt.event.ActionEvent) {
    dbViewer.update();// prod the DB viewer.
}
```

4.b) In order to make the event hookup at run-time they also have to add a line to their MyApplet.init method:

```
btn.addActionListener(this);
```

5. Packaging up the applet

5.a) The user decides they are done. They compile their MyApplet.java file and make sure that the MyApplet.class file gets added to their class directory.

5.b) They create a test HTML page that uses the applet tag to reference MyApplet with a suitable codebase.

6. Trying it out

6.a) The user then tests out the applet in their SpaceWars web browser. When they visit the html page, the applet is initialized by running its MyApplet.init method.

6.b) The MyApplet.init method does a "new" of a Button bean and uses Beans.instantiate to read in an initialized DatabaseViewer bean.

6.c) The init method then calls various setXXX methods to customize the beans and then the button registerActionListener to hookup the button events.

6.d) The result is a running applet with a working event hookup from the button to the database viewer.

6.e) Restrained Joy, Limited Happiness, and decaffe coffee all round.

Here's the complete sample code for the hand-written applet:

```
class MyApplet extends java.awt.Applet implements
                                    java.awt.event.ActionListener {

     // Applet initialization.
     public void init() {
          btn = new dave.Button();
          add(btn);
          try {
               ClassLoader cl = Class.forName(
                              "dave.Button").getClassLoader();
               dbViewer = Beans.instantiate(cl,
                                   "dave/DatabaseViewer.ser");
               add(dbViewer);
          } catch (Exception ex) {
               throw new Error("Couldn't make dbViewer");
          }

          btn.setBackground(java.awt.Color.Red);
          dbViewer.setURL("jdbc:msql://aardvark.eng/stuff");

          // Hookup the button to fire action events at us.
          btn.addActionListener(this);
     }

     // Method that catches action events from the button.
     void actionPerformed(java.awt.event.ActionEvent evt) {
          dbViewer.update();    // prod the DB viewer.
     }

     private DatabaseViewer dbViewer;
     private Button btn;
}
```

4 Windowing Environment Issues

A number of new features are being added to AWT in JDK 1.1 These features are described in the “JDK 1.1- AWT Enhancements” web pages (see “Related Documents” on page 6). However because some features are likely to be important to beans developers, we provide a short overview of key new AWT features in this section.

4.1 Uniform Data Transfer mechanism

The uniform data transfer mechanism proposed for inclusion into AWT is intended to provide the fundamental mechanisms to enable the interchange of structured data between objects, applets or applications using higher level, user oriented facilities such as Cut, Copy and Paste, or Drag and Drop. In particular, the UDT mechanisms design goals are:

- provide a simple conceptual framework for the encapsulation and interchange of data.
- provide a general API that can support current and future data interchange models.
- enable the efficient, dynamic creation, registration, and interchange of rich and diverse data types.
- enable data interchange across process boundaries.
- enable the interchange of data between Java-based and platform-native applications.

The facilities are based around an interface, *Transferable,* that defines abstractions both for the transferable data itself and a namespace, based on the class *DataFlavor,* for identifying and negotiating the type, or types, of data that may be exchanged.

Objects wishing to provide a structured representation of some aspect of the information they encapsulate will implement the *Transferable* interface, and possibly introduce additional data types identifying new types of structured data by creating new *DataFlavor*s.

Objects wishing to obtain such structured data will obtain references to a particular *Transferable* interface and will compare the *DataFlavor*s available to determine if the source offers a type, or types, that are familiar to the destination.

The current proposal, at the time of writing, is that the MIME namespace for data types shall be used by *DataFlavors*.

4.2 Clipboard/Selection services

Clipboard, or Selection services build upon the abstractions provided by the Uniform Data Transfer mechanisms to enable objects, applets, or applications to exchange structured data between each other, by introducing the concept of a *Clipboard* class*,* or named Selection, that provides a rendezvous for the interchange between the source and destination, and a *ClipboardOwner* interface that provides data sources with notifications of clipboard state changes.

4.3 Drag and Drop

Drag and Drop also builds upon the Uniform Data Transfer model to provide abstractions that enable the accelerated interchange of data through the user model of the Drag and Drop gesture.

The design goals of this API are to:

- provide a simple set of abstractions and APIs
- provide fully featured Drag and Drop behavior on all platforms.

The API consists of four specifications:

- *DragSource*, an interface which encapsulates the behaviors of objects that wish to initiate Drag and Drop gestures.
- *DragContext*, a class which encapsulates the state of a Drag for its duration.
- *DropTarget*, an interface which encapsulates the behaviors of objects that wish to participate in Drag, and any subsequent Drop, should one occur.
- *DropContext*, a class that encapsulates the state of a drop operation.

5 Persistent Storage

5.1 Goals

Java Beans need to support a very wide range of storage behaviour. We want them to be able to use existing data formats and plug into OLE or OpenDoc documents. For example, we want them to be able to pretend to be an Excel document inside a Word document.

We'd also like it to be "trivial" for the common case of a tiny Java Bean that simply wants to have its internal state saved and doesn't want to think about it.

If Java Beans use the "native" Java format then their persistent state should be portable between different platforms.

Not all Java Beans need have persistent state. However typically those beans that allow their properties to be edited will provide a way for storing away and reloading the changes.

5.2 A composite solution

To meet these different goals we will support a composite solution where a Java Bean can either choose to use the automatic Java serialization mechanism or where it can choose to use an "externalization" stream mechanism that provides it with full control over the resulting data layout.

As part of Java Beans 1.0 we support the Java Object Serialization mechanism which provides an automatic way of storing out and restoring the internal state of a collection of Java objects. For further information see the Java Serialization specification. (See "Related Documents" on page 6.)

The "externalization" mechanism is under active design and will be included in a future version of the Beans APIs. It will allow Java objects complete control over the writing of their state, so that they can chose to mimic arbitrary existing data formats, and will also support structured storage.

5.3 What should be saved

When a bean is made persistent it should store away appropriate parts of its internal state so that it can be resurrected later with a similar appearance and similar behaviour. Normally a bean will store away persistent state for all its exposed properties. It may also store away additional internal state that is not directly accessible via properties. This might include (for example) additional design choices that were made while running a bean Customizer (see Section 5) or internal state that was created by the bean developer.

However a bean should not normally store away pointers to external beans (either peers or a parent container) but should rather expect these connections to be rebuilt by higher-level software. So normally it should use the "transient" keyword to mark pointers to other beans or to event listeners. In general it is a container's responsibility to keep track of any inter-bean wiring it creates and to store and resurrect it as needed.

For the same reasons, normally event adaptors should mark their internal fields as "transient".

All application builder tools for use with Java Beans must support the use of Java serialization as a way of storing beans internal state. However builder tools may use their own mechanisms for keeping track of connections between beans, and for keeping track of event hookups.

5.4 Deserialization and classpaths

As part of JDK 1.1 it will be possible to deserialize an object from a .ser file associated with a class-loader. When this happens, the class-loader used for the .ser file will also be used as the default class-loader for any .class files that needed to be loaded while deserializing the object.

Thus you will be able to deliver a serialized object and a set of .class files as a single unit, and the deserialized object can use the associated .class files.

6 Events

Events are one of the core features of the Java Beans architecture. Events provide a convenient mechanism for allowing components to be plugged together in an application builder, by allowing some components to act as sources for event notifications that can then be caught and processed by either scripting environments or by other components.

6.1 Goals

Conceptually, events are a mechanism for propagating state change notifications between a *source* object and one or more target *listener* objects.

Events have many different uses, but a common example is their use in window system toolkits for delivering notifications of mouse actions, widget updates, keyboard actions, etc.

For Java, and Java Beans in particular, we require a generic, extensible event mechanism, that:

1. Provides a common framework for the definition and application of an extensible set of event types and propagation models, suitable for a broad range of applications.
2. Integrates well with, and leverages the strengths of, the features of the Java language and environment.
3. Allows events to be caught (and fired) by scripting environments.
4. Facilitates the use of application builder technology to directly manipulate events and the relationships between event sources and event listeners at design time.
5. Does not rely on the use of sophisticated application development tools.

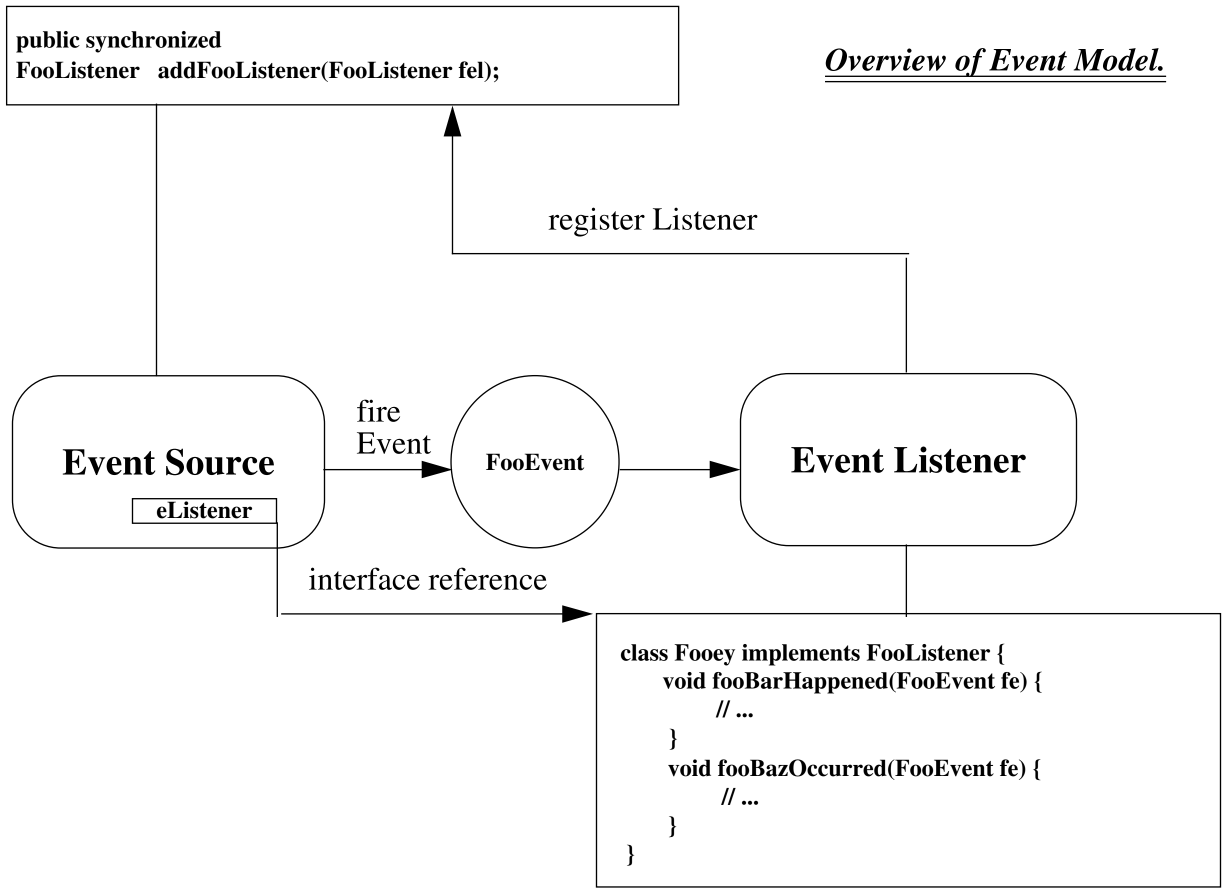
In particular, the design should:

6. Enable the discovery of the events that a particular object's class may generate.
7. Enable the discovery of the events that a particular object's class may observe.
8. Provide a common event registration mechanism that permits the dynamic manipulation of the relationships between event sources and event listeners.
9. Be implementable without requiring changes to the VM or the language.
10. Promote high performance propagation of events between sources and listeners.
11. Enable a natural mapping between Java Beans events and event models associated with other common component architectures.

6.2 Overview

The details of the event model are described in the following sections, but in outline:

- Event notifications are propagated from sources to listeners by Java method invocations on the target listener objects.
- Each distinct kind of event notification is defined as a distinct Java method. These methods are then grouped in *EventListener* interfaces that inherit from *java.util.EventListener.*



- Event listener classes identify themselves as interested in a particular set of events by implementing some set of *EventListener* interfaces.
- The state associated with an event notification is normally encapsulated in an event state object that inherits from *java.util.EventObject* and which is passed as the sole argument to the event method.
- Event sources identify themselves as sourcing particular events by defining registration methods that conform to a specific design pattern (see Sections 8.2 and 8.4) and accept references to instances of particular *EventListener* interfaces.
- In circumstances where listeners cannot directly implement a particular interface, or when some additional behavior is required, an instance of a custom adaptor class may be interposed between a source and one or more listeners in order to establish the relationship or to augment behavior.

6.3 Event State Objects

Information associated with a particular event notification is normally encapsulated in an “event state” object that is a subclass of *java.util.EventObject.* By convention these event state classes are given names ending in “Event”. For example:

```
public class MouseMovedExampleEvent extends java.util.EventObject {
     protected int x, y;

     // constructs a new MouseMovedExampleEvent
     MouseMovedExampleEvent(java.awt.Component source, Point location) {
          super(source);
          x = location.x;
          y = location.y;
     }

     // Access method for location
     public Point getLocation() {
          return new Point(x, y);
     }

     // translates coords, for use when propagating up view hierarchy.
     public void translateLocation(int dx, int dy) {
          x += dx;
          y += dy;
     }
}
```

As with *java.lang.Exception*, new subclasses of *java.util.EventObject* may be created simply to allow logical distinctions between event state objects of different types, even if they share all the same data. So another example might be:

```
public class ControlExampleEvent extends java.util.EventObject {
     // Simple "logical" event. The significant information is the type
     // of the event subclass.
     ControlExampleEvent(Control source) {
          super(source);
     }
}
```

Typically, event state objects should be considered immutable. Therefore it is strongly recommended that direct public access to fields be denied, and that accessor methods be used to expose details of event state objects. However where some aspects of an event state object require modification (e.g translating view relative coordinates when propagating an event through a view hierarchy, as in the example above) it is recommended that such modifications be encapsulated by an appropriate method that effects the required modification (as in the sample code above), or alternatively a new event state object should be instantiated reflecting the appropriate modifications.

These accessor methods should follow the appropriate design patterns for read-only, read/write, or write-only properties as defined in Section 8.3. This is especially important as it enables the framework to identify in-out event data when bridging events between Java and other component architectures that encompass the ability to return state to the originator of an event, through the event state object.

6.4 EventListener Interfaces

Since the new Java event model is based on method invocation we need a way of defining and grouping event handling methods. We require that event handling methods be defined in *EventListener* interfaces that inherit from *java.util.EventListener*. By convention these *EventListener* interfaces are given names ending in “Listener”.

A class that wants to handle any of the set of events defined in a given *EventListener* interface should implement that interface.

So for example one might have:

```
public class MouseMovedExampleEvent extends java.util.EventObject {
     // This class defines the state object associated with the event
     ...
}

interface MouseMovedExampleListener extends java.util.EventListener {
     // This interface defines the listener methods that any event
     // listeners for “MouseMovedExample” events must support.
     void mouseMoved(MouseMovedExampleEvent mme);
}

class ArbitraryObject implements MouseMovedExampleListener {
     public void mouseMoved(MouseMovedExampleEvent mme) {
          ...
     }
}
```

Event handling methods defined in *EventListener* interfaces should normally conform to a standard design pattern. Requiring conformance to such a pattern aids in the utility and documentation of the event system, permits such interfaces to be determined by third parties programmatically, and allows the automatic construction of generic event adaptors.

The signature of this design pattern is:

```
void <eventOccurenceMethodName>(<EventStateObjectType> evt);
```

Where the <EventStateObjectType> is a subclass of *java.util.EventObject*.

It is also permissible for the event handling method to throw “checked” exceptions, which should be declared in the normal way in a throws clause in the method signature. (See also Section 6.6.3).

Typically, related event handling methods will be grouped into the same *EventListener* interface. So for example, mouseEntered, mouseMoved, and mouseExited might be grouped in the same *EventListener* interface. (However some *EventListener* interfaces may only contain a single method.) In addition, in situations where there are a large number of related kinds of events, a hierarchy of *EventListener*s may be defined such that the occurrences most commonly of interest to the majority of potential listeners are exposed in their own *EventListener* interface, and the more complete set of occurrences are specified in subsequent *EventListener* interfaces perhaps as part of an inheritance graph, thus reducing the implementation burden for the simple case. For example:

```
public class ControlEvent extends java.util.EventObject {
     // ...
}

interface ControlListener extends java.util.EventListener {
     void controlFired(ControlEvent ce);
}

interface ComplexControlListener extends ControlListener {
     void controlHighlighted(ControlEvent ce);
     void controlPreviewAction(ControlEvent ce);
     void controlUnhighlighted(ControlEvent ce);
     void controlHelpRequested(ControlEvent ce);
}
```

6.4.1 Event methods with arbitrary argument lists

As described above, normally event handling methods should have a single argument which is a sub-type of *java.util.EventObject*.

However under certain specific and unusual applications or circumstances, conformance to this standard method signature may not be appropriate. This may happen, for example, when forwarding event notifications to external environments that are implemented in different languages or with different conventions. In these circumstances it may be necessary to express the notification method signature in a style more in keeping with the target environment.

In these exceptional circumstances it is permissible to allow method signatures containing one or more parameters of any arbitrary Java type.

The permitted signature is:

```
void <event_occurence_method_name>(<arbitrary-parameter-list>);
```

The method may also have a “throws” clause listing arbitrary exceptions.

Designers are strongly recommended to use considerable restraint in the application of this arbitrary argument list in the design of their listener interfaces as there are several drawbacks to its widespread application. Therefore it is recommended that its usage be confined to situations where solutions utilizing the recommended pattern in conjunction with custom event adaptors are not generally applicable, as in the examples mentioned above.

Although usage of this relaxation in the design pattern is strongly discouraged for general application, there will be circumstances where it will be used to good effect. Applications builder environments should support such relaxed event method signatures and in general should not discriminate between such methods and those that strictly conform to the standard design pattern.

6.5 Event Listener Registration

In order for potential *EventListeners* to register themselves with appropriate event source instances, thus establishing an event flow from the source to the listener, event source classes must provide methods for registering and de-registering event listeners.

The registration methods shall conform to a particular set of design patterns that aids in documentation and also enables programmatic introspection by the Java Beans introspection APIs and by application builder tools.

The standard design pattern for EventListener registration is:

```
public void add<ListenerType>(<ListenerType> listener);
public void remove<ListenerType>(<ListenerType> listener);
```

The presence of this pattern identifies the implementor as a standard multicast event source for the listener interface specified by *<ListenerType>*.

Invoking the add*<ListenerType>* method adds the given listener to the set of event listeners registered for events associated with the *<ListenerType>*. Similarly invoking the remove*<ListenerType>* method removes the given listener from the set of event listeners registered for events associated with the *<ListenerType>*.

> The add*<ListenerType>* and remove*<ListenerType>* methods should normally be *synchronized* methods to avoid races in multi-threaded code.

The relationship between registration order and event delivery order is implementation defined. In addition the effects of adding the same event listener object more than once on the same event source, or of removing an event listener object more than once, or of removing an event listener object that is not registered, are all implementation defined.

6.5.1 Event Registration Example

An example of event listener registration at a normal multicast event source:

```
public interface ModelChangedListener extends java.util.EventListener {
     void modelChanged(EventObject e);
}

public abstract class Model {
     private Vector listeners = new Vector(); // list of Listeners

     public synchronized void
     addModelChangedListener(ModelChangedListener mcl) {
          listeners.addElement(mcl);
     }

     public synchronized void
     removeModelChangedListener(ModelChangedListener mcl) {
          listeners.removeElement(mcl);
     }
```

```
        protected void notifyModelChanged() {
             Vector    l;
             EventObject e = new EventObject(this);

             // Must copy the Vector here in order to freeze the state of the
             // set of EventListeners the event should be delivered to prior
             // to delivery. This ensures that any changes made to the Vector
             // from a target listener's method, during the delivery of this
             // event will not take effect until after the event is delivered

             synchronized(this) { l = (Vector)listeners.clone(); }

             for (int i = 0; i < l.size(); i++) { // deliver it!
                  ((ModelChangedListener)l.elementAt(i)).modelChanged(e);
             }
        }
   }
```

6.5.2 Unicast Event Listener Registration.

Wherever possible, event sources should support multicast event notification to arbitrary collections of event listeners. However, for either semantic or implementation reasons it may sometimes be inappropriate or impractical for some events to be multicast. We therefore also permit a unicast design pattern for event registration.

The type signature for a unicast EventListener registration pattern is:

```
public void add<ListenerType>(<ListenerType> listener)
                         throws java.util.TooManyListenersException;
public void remove<ListenerType>(<ListenerType> listener);
```

The presence of the `throws java.util.TooManyListenersException` in the pattern identifies the implementor as an event source with unicast event source semantics for the interface specified by *<ListenerType>*.

Note that the unicast pattern is a special case of the multicast pattern and as such allows migration of a unicast event source to a multicast event source, without breaking existing client code

Invoking the add*<ListenerType>* method on a unicast source registers the specified *<ListenerType>* instance as the current listener only if no other listener is currently registered, otherwise the value of the currently registered listener is unchanged and the method throws:

```
java.util.TooManyListenersException.
```

Passing *null* as the value of the listener is illegal, and may result in either an *IllegalArgumentException* or a *NullPointerException*.

6.6 Event Delivery Semantics

6.6.1 Unicast/Multicast

Normal event delivery is multicast. Each multicast event source is required to keep track of a set of event listeners for each different kind of event it fires. When an event is triggered, the event source should call each target listener. However if any of the event listeners raise an exception (see 6.6.3 below) it is an implementation decision for the event source whether to continue and deliver the event to the remaining listeners.

As described in Section 6.5.2 some event sources may only support unicast. In this case the event source is only required to keep track of a single event listener for each different unicast event it fires. When a unicast event is triggered, the event source should call this single target listener.

6.6.2 Synchronous delivery

Event delivery is synchronous with respect to the event source. When the event source calls a target listener method, the call is a normal Java method invocation and is executed synchronously, using the caller’s thread.

6.6.3 Exceptional Conditions.

Target listener methods are permitted to throw checked Exceptions. It is an implementation decision for the event source object how to react when a listener method has raised an exception.

In general, listener methods should not throw unchecked exceptions (such as NullPointerException), as these cannot be documented in throws clauses. However event sources should be aware that target listeners may inadvertently throw such unchecked exceptions. Normally this should be treated as a bug in the target listener.

6.6.4 Concurrency control

In multi-threaded systems event handling is a common cause of both races and deadlocks. This is partly due to the inherently asynchronous nature of event handling and partly because event handling often flows “upwards” against the normal program control flow, so that component A may be delivering an event to component B at the same moment as component B is performing a normal method invocation into component A.

We recommend that event sources should use synchronized methods and synchronized blocks to synchronize access to the data structures that describe event listeners.

In order to reduce the risk of deadlocks, we strongly recommend that event sources should avoid holding their own internal locks when they call event listener methods. Specifically, as in the example code in Section 6.5.1, we recommend they should avoid using a synchronized method to fire an event and should instead merely use a synchronized block to locate the target listeners and then call the event listeners from unsynchronized code.

6.6.5 Modifications to the Set of Event Listeners during Event Delivery.

While an event is being multicast it is possible that the set of event listeners may be updated, either by one of the target event listener methods, or by some other thread.

The exact effects of such concurrent updates are implementation specific. Some event sources may consult the currently registered listener set as they proceed and only deliver the event to listeners still on the current list. Other implementations may chose to make a copy of the event target list when they start the event delivery and then deliver the event to that exact set.

Note that this means that an event listener may be removed from an event source and then still receive subsequent event method calls from that source, because there were multicast events in progress when it was removed.

6.7 Event Adaptors

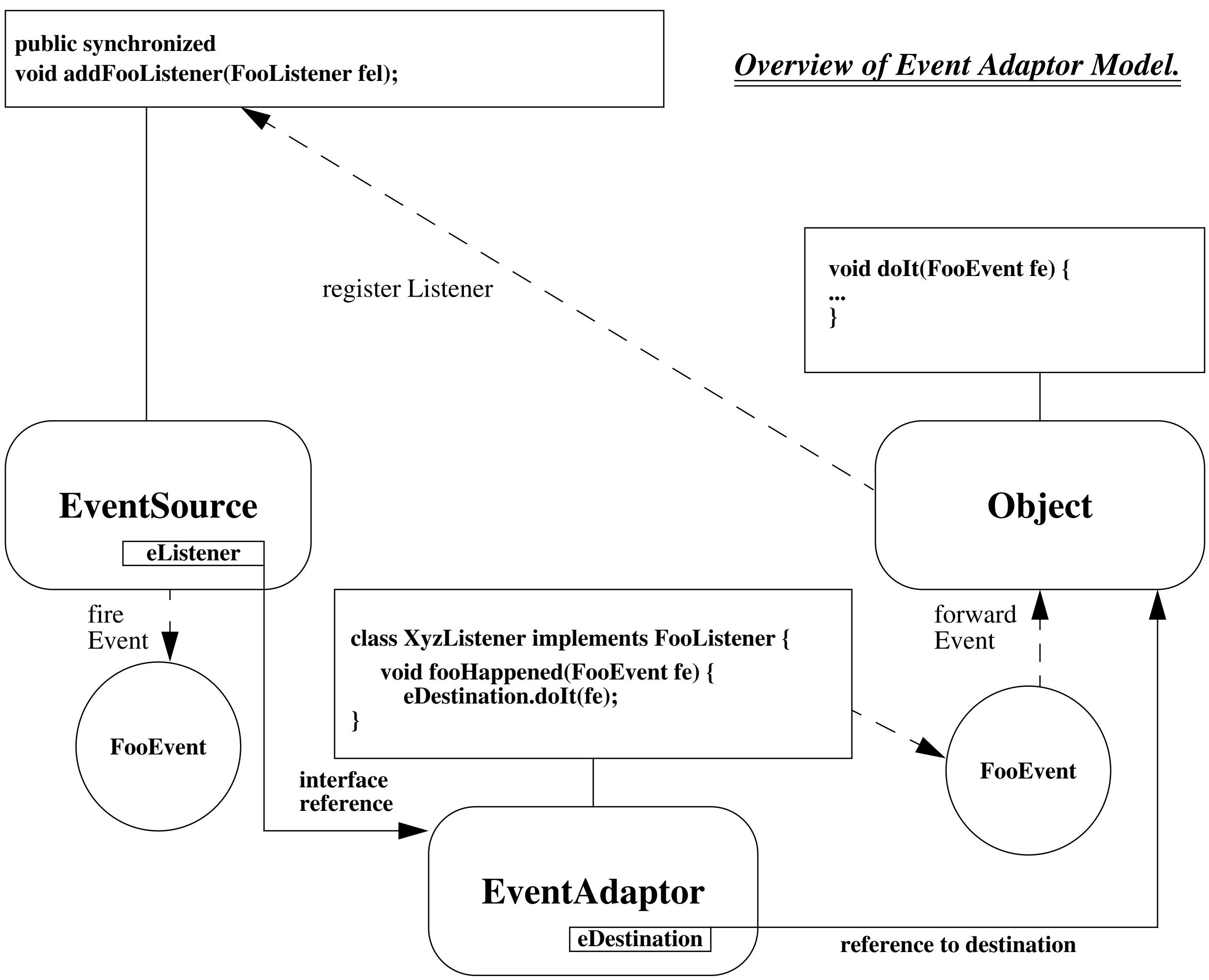
Event adaptors are an extremely important part of the Java event model.

Particular applications or application builder tools may chose to use a standard set of event adaptors to interpose between event sources and event listeners to provide additional policy on event delivery .

6.7.1 Event Adaptor Overview

When additional behavior is required during event delivery, an intermediary "event adaptor" class may be defined, and interposed between an event source and the real event listener.

The primary role of an event adaptor is to conform to the particular *EventListener* interface expected by the event source, and to decouple the incoming event notifications on the interface from the actual listener(s).



Some examples of the uses of such adaptors are

- Implementing an event queuing mechanism between sources and listeners.
- Acting as a filter.
- Demultiplexing multiple event sources onto a single event listener.
- Acting as a generic "wiring manager" between sources and listeners.

6.7.2 Security issues

Event adaptors that simply run as parts of downloaded applets add no new security issues. They will be constrained by the standard applet security model.

However event adaptors that are added to the standard base system must be careful to enforce the current Java security model. Many security checks within the current Java virtual machine are enforced by scanning backwards through the current thread's stack to see if any stack frame belongs to an untrusted applet and if so denying access. Because the current event mechanism delivers an event synchronously within the firing thread, this stack checking algorithm means that actions by an event recipient are automatically limited if the event source was an untrusted applet. This protects against certain security attacks where an applet fires bogus events to mislead a component.

Any event adaptors that are potentially going to be installed on the local disk and which may be used by downloaded applets must be careful to maintain this model. Thus an event-queueing adaptor must be careful that its event delivery threads do not look more trustworthy than the original event firer.

The only event adaptors that JavaSoft currently intends to deliver as part of the base JDK are some simple utility base classes to make it easier to implement AWT event listeners. These adaptors do not introduce any new security issues.

6.7.3 Demultiplexing Event Adaptors.

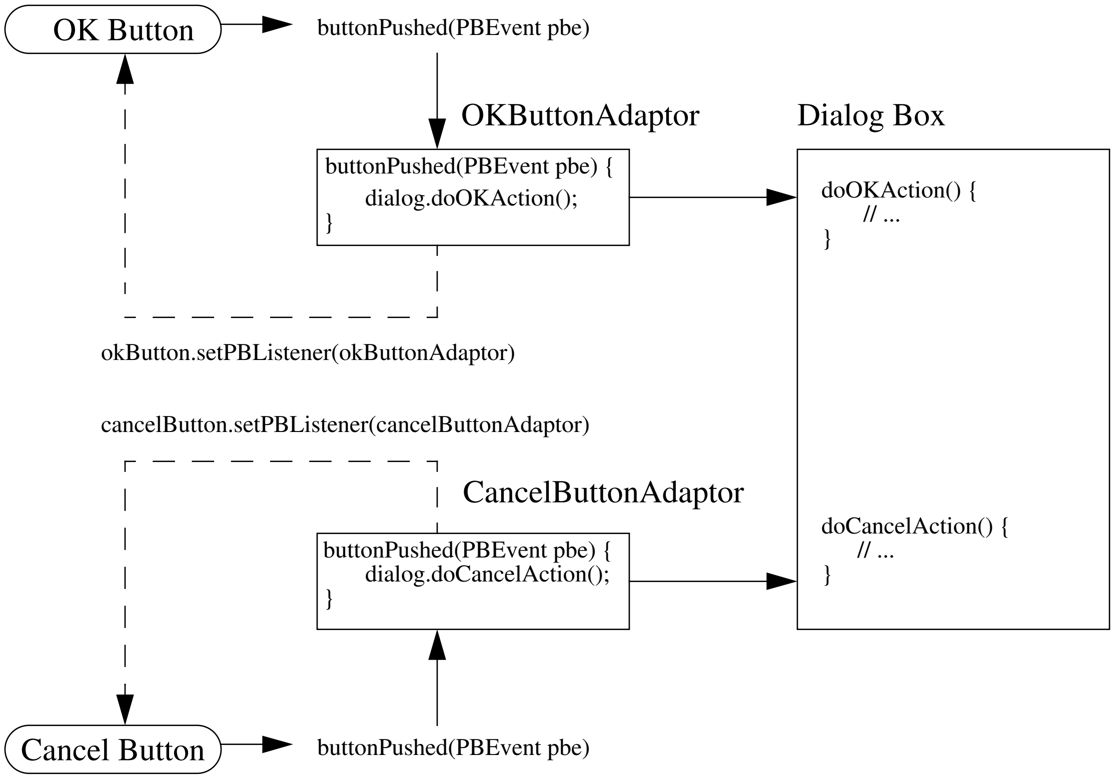
Demultiplexing is perhaps the most common expected use of the adaptor technique.

Any given event listener object may only implement a given *EventListener* interface once. Thus if a listener registers itself with multiple event sources for the same event, the listener has to determine for itself which source actually emitted a particular event.

To make event delivery more convenient, we would like to allow the events for different event sources to be delivered to different target methods on the final listener object. So for example if I have two buttons "OKButton" and "CancelButton", I would like to be able to arrange that a push on the "OKButton" results in a call on one method, whereas a push on the "CancelButton" results in a call on another method.

We can solve this problem by interposing "demultiplexing" adaptors between each source and the listener, where each demultiplexing adaptor takes an incoming method call of one name and then calls a differently named method on its target object.

In the example (see the diagram and code below) a DialogBox object has two push buttons "OK" and "Cancel", both of which fire a *buttonPushed(PBEvent)* method. The DialogBox is



designed to invoke the methods, *doOKAction()* when the "OK" button fires, and *doCancelAction()* when the "Cancel" button fires.

The DialogBox defines two classes, *OKButtonAdaptor* and *CancelButtonAdaptor* that both implement the *PBListener* interface but dispatch the incoming notification to their respective action methods.

As a side effect of instantiation of the DialogBox, instances of the private adaptors are also created and registered with the *PushButton* instances, resulting in the appropriate event flow and mapping.

```
// Adaptor to map "Cancel Pressed" events onto doCancelAction()

class CancelAdaptor implements PushButtonExampleListener {
     private Dialog dialog;
     public CancelAdaptor(Dialog dest) {
          dialog = dest;
     }
     public void PushButtonPressed(PushButtonExampleEvent pbe) {
          dialog.doCancelAction();
     }
}
```

```
// Adaptor to map "OK Pressed" events onto doOKAction()

class OKAdaptor implements PushButtonExampleListener {
     private Dialog dialog;
     public OKAdaptor(Dialog dest) {
          dialog = dest;
     }
     public void PushButtonPressed(PushButtonExampleEvent pbe) {
          dialog.doOKAction();
     }
}

// Define generic Dialog with two methods, doOKAction()
// and doCancelAction()

public abstract class DialogBox{
     // Dialog with two adaptors for OK and Cancel buttons
     protected PushButton okButton;
     protected PushButton cancelButton;
     protected OKAdaptor okAdaptor;
     protected CancelAdaptor cancelAdaptor;

     DialogBox() {
          okButton  = new PushButton("OK");
          cancelButton= new PushButton("Cancel");

          okAdaptor   = new OKAdaptor(this);
          okButton.addPushButtonExampleListener(OKAdaptor)

          cancelAdaptor= new CancelAdaptor(cancelButton, this);
          cancelButton.addPushButtonExampleListener(CancelAdaptor);
     }

     public abstract void doOKAction();
     public abstract void doCancelAction();
}
```

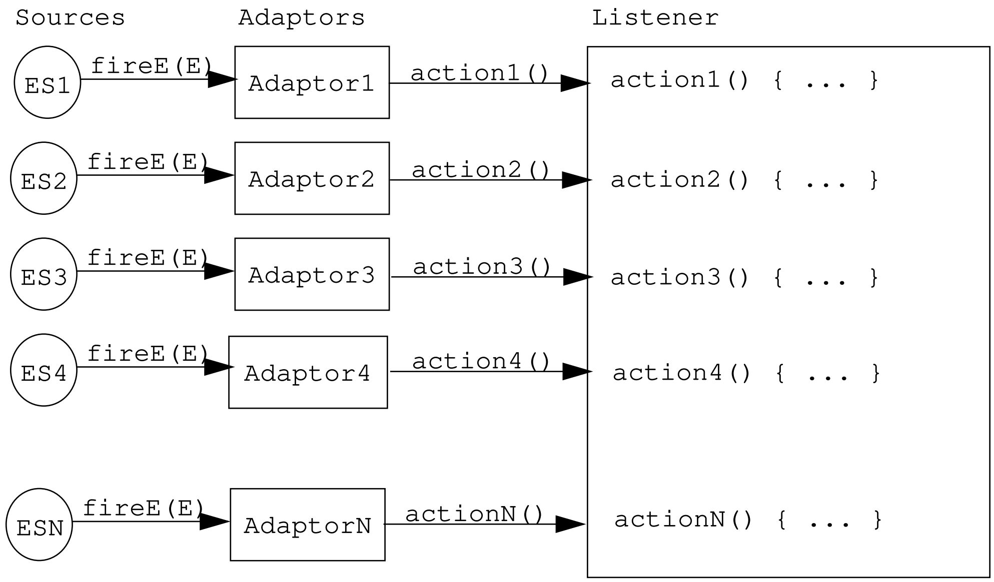
6.7.4 Generic demultiplexing adaptors.

The simple demultiplexing adaptor model described in Section 6.7.3 has the drawback that the number of private adaptor classes can become very high if there are large number of different event targets, as a different adaptor class is required for each source-target hookup.

Where there are a large number of event source instances another technique can be employed in order to obviate the need for an adaptor class per source-target hookup.

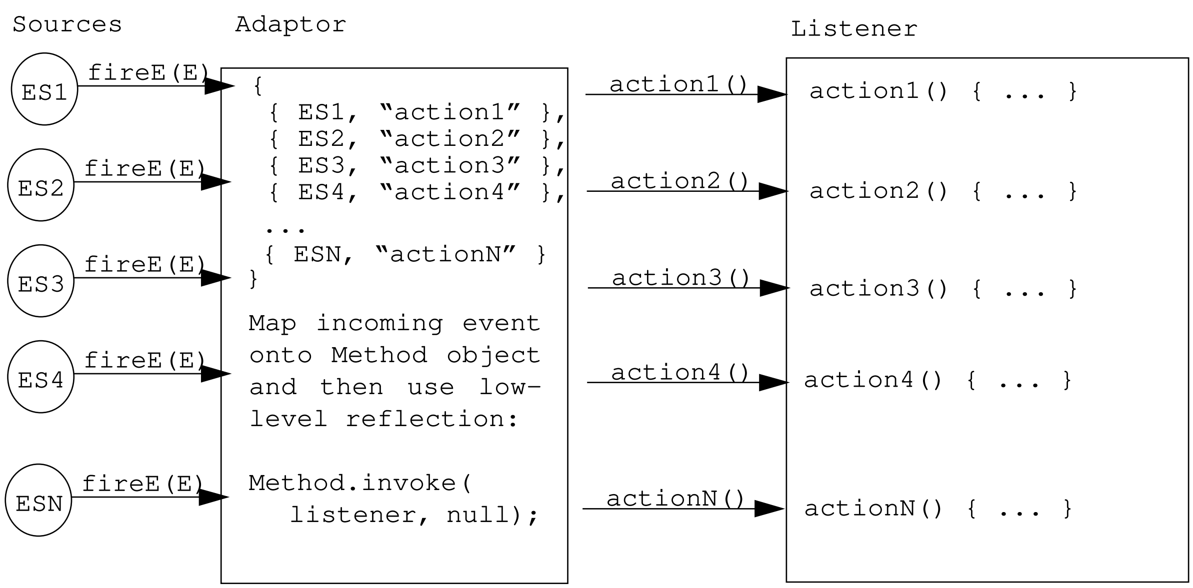
This technique uses generic adaptors and is most suited to automatic code generation by application builder tools. It involves synthesizing an adaptor class that utilizes the new low-level Java introspection APIs to map incoming events from multiple source instances onto particular method invocations via the *invoke* method of the *java.lang.reflect.Method* class.

The shortcoming of this technique however is that the use of the low-level introspection APIs for method lookup and invocation via string names is in some senses "unnatural", and it defeats the strong typing facilities.



Adaptor Proliferation with Simple Multiplexing Adaptors.

Therefore it is not recommended that this technique be used in manually generated code as it may lead to fatal errors in an application using this technique at runtime. It is instead recommended that this technique only be used by application builder generated code, where greater constraints may be applied to, and subsequently checked during, the code generation and thus there is less risk of introducing serious runtime errors into application code using this technique.



Using a Generic Multiplexing Adaptor.

Interface EventListener

```
public interface java.util.EventListener
```

The EventListener interface is the prototypical interface from which all event listener interfaces shall be derived. EventListener defines no methods and exists merely as a way of identifying sub-types as event listeners.

By convention all derived EventListener interfaces shall specify their event notification methods according to a recognisable design pattern with a method signature of:

```
public void eventNotificationMethodName(EventType e);
```

Class EventObject

```
public class java.util.EventObject
    extends java.lang.Object
```

The Event class is the abstract root class from which all event state objects shall be derived.

All Event's are constructed with a reference to the object, the "source", that is logically deemed to be the object upon which the Event in question initially occurred upon.

Constructors

- **EventObject**

  `public EventObject(Object source)`

  Constructs a prototypical Event

  Parameters:

  source
  The object that the Event occurred upon.

Methods

- **getSource**

  `public Object getSource()`

  Returns:

  The object that the Event initially occurred upon.

7 Properties

Properties are discrete, named attributes of a Java Bean that can affect its appearance or its behaviour. For example, a GUI button might have a property named “Label” that represents the text displayed in the button.

Properties show up in a number of ways:

1. Properties may be exposed in scripting environments as though they were fields of objects. So in a Javascript environment I might do “b.Label = foo” to set the value of a property.
2. Properties can be accessed programmatically by other components calling their getter and setter methods (see Section 7.1 below).
3. As part of the process of customizing a component (see Section 9), its properties may be presented in a property sheet for a user to edit.
4. Typically a bean’s properties will be persistent, so that their state will be stored away as part of the persistent state of the bean.

Properties can have arbitrary types, including both built-in Java types such as “int” and class or interfaces types such as “java.awt.Color”.

7.1 Accessor methods

Properties are always accessed via method calls on their owning object. For readable properties there will be a *getter* method to read the property value. For writable properties there will be a *setter* method to allow the property value to be updated. Thus even when a script writer types in something such as “b.Label = foo” there is still a method call into the target object to set the property, and the target object has full programmatic control.

So properties need not just be simple data fields, they can actually be computed values. Updates may have various programmatic side effects. For example, changing a bean’s background color property might also cause the bean to be repainted with the new color.

For simple properties the accessor type signatures are:

```
void setFoo(PropertyType value); // simple setter
PropertyType getFoo();           // simple getter
```

GetFoo and *setFoo* are simply example names. Accessor methods can have arbitrary names. However for standard naming conventions for accessor methods see the design patterns described in Section 8.3.

7.2 Indexed properties

In addition to simple single-value properties, we also support *indexed properties*. An indexed property supports a range of values. Whenever the property is read or written you just specify an index to identify which value you want.

Property indexes must be Java “int”s.

We may relax this restriction in the future to allow other index types. However it seems that "int" indexes are in practice the most common and useful kind of property index.

A component may also expose an indexed property as a single array value. For example, if there is an indexed property "fred" of type string it may be possible from a scripting environment to access an individual indexed value using "b.fred[3]" and also to access the same property as an array using "b.fred".

For indexed properties the accessor type signatures are:

```
void setter(int index, PropertyType value);  // indexed setter
PropertyType getter(int index);              // indexed getter
void setter(PropertyType values[]);          // array setter
PropertyType[] getter();                     // array getter
```

The indexed getter and setter methods may throw a *java.lang.ArrayIndexOutOfBoundsException* runtime exception if an index is used that is outside the current array bounds.

In order to change the size of the array you must use the array setter method to set a new (or updated) array.

7.3 Exceptions on accessor methods

Both simple and indexed property accessor methods may throw checked exceptions. This allows property setter/getter methods to report exceptional conditions.

Thus for example you might have:

```
void setWombat(Wombat value) throws BadWombatException;
Marsupial getKangaroo(int index) throws AwfulCommunicationException;
```

7.4 Bound and constrained properties

Java Beans provides support for *bound* and *constrained* properties. These are advanced features that are not required for simple beans or for initial beans programming.

7.4.1 Bound properties

Sometimes when a bean property changes then either the bean's container or some other bean may wish to be notified of the change.

A component can chose to provide a change notification service for some or all of its properties. Such properties are commonly known as *bound properties*, as they allow other components to bind special behaviour to property changes.

The *PropertyChangeListener* event listener interface is used to report updates to simple bound properties. If a bean supports bound properties then it should support a normal pair of multicast event listener registration methods for PropertyChangeListeners:

```
public void addPropertyChangeListener(PropertyChangeListener x);
public void removePropertyChangeListener(PropertyChangeListener x);
```

When a property change occurs on a bound property the bean should call the PropertyChangeListener.propertyChange method on any registered listeners, passing a *PropertyChangeEvent* object that encapsulates the locale-independent name of the property and its old and new values.

The event source should fire the event *after* updating its internal state.

For programming convenience, we provide a utility class PropertyChangeSupport that can be used to keep track of PropertyChangeListeners and to fire PropertyChange events.

7.4.2 Constrained properties

Sometimes when a property change occurs some other bean may wish to validate the change and reject it if it is inappropriate.

We refer to properties that undergo this kind of checking as *constrained properties*.

In Java Beans, constrained property setter methods are required to support the *PropertyVetoException*. This documents to the users of the constrained property that attempted updates may be vetoed.

So a simple constrained property might look like:

```
PropertyType getFoo();
void setFoo(PropertyType value) throws PropertyVetoException;
```

The *VetoableChangeListener* event listener interface is used to report updates to constrained properties. If a bean supports constrained properties then it should support a normal pair of multicast event listener registration methods for VetoableChangeListeners:

```
public void addVetoableChangeListener(VetoableChangeListener x);
public void removeVetoableChangeListener(VetoableChangeListener x);
```

When a property change occurs on a constrained property the bean should call the VetoableChangeListener.vetoableChange method on any registered listeners, passing a *PropertyChangeEvent* object that encapsulates the locale-independent name of the property and its old and new values. If the event recipient does not wish the requested edit to be performed it may throw a PropertyVetoException. It is the source bean's responsibility to catch this exception, revert to the old value, and issue a new VetoableChangeListener.vetoableChange event to report the reversion.

> The initial vetoableChangeListener.vetoablePropertyChange event may have been multicast to a number of recipients before one vetoes the new value. If one of the recipients vetoes, then we have to make sure that all the other recipients are informed that we have reverted to the old value. Thus the need to fire another VetoableChangeListener.vetoableChange event when we revert. The source may chose to ignore vetoes when reverting to the old value.

The event source should fire the event *before* updating its internal state.

For programming convenience we provide a utility class VetoableChangeSupport that can be used to keep track of VetoableChangeListeners and to fire VetoableChange events. It also catches PropertyVetoExceptions and sends any necessary reversion event.

7.4.3 Listening for both bound and constrained properties

If a bean supports a property that is both bound and constrained, then it should fire a VetoableChangeListener.vetoableChange event before updating the property and a PropertyChangeListener.propertyChange event after updating the property.

7.4.4 Checking for changes that have no effect

If a property setter method gets called with an argument that is equal to the property's current value, then we recommend that the bean does not fire either a VetoableChangeListener.vetoableChange event or a PropertyChangeListener.propertyChange event. This behaviour is both more efficient and reduces the danger of loops when property setter methods are cross connected.

Thus for example the following property update on a bound property "fred" need not fire a PropertyChangeEvent:

```
x.setFred(x.getFred());
```

For code that uses the PropertyChangeSupport and VetoableChangeSupport utility classes, this check for equality is handled automatically, as these classes check if the "old" and "new" property values are equal and if so avoid firing an event.

7.4.5 Optional event handlers for specific properties

If a bean supports bound properties then it must allow a global PropertyChangeListener to be registered using the addPropertyChangeListener method. Similarly, if it supports constrained properties then it must allow a global VetoableChangeListener to be registered using the addVetoableChangeListener method.

However, in addition to this, a bean may optionally allow either PropertyChangeListeners or VetoableChangeListeners to be registered on a per-property basis as an optimization. This is entirely optional, but if a bean chooses to support this then for each such bound property it should provide methods of the form:

```
void add<PropertyName>Listener(PropertyChangeListener p);
void remove<PropertyName>Listener(PropertyChangeListener p);
```

and it can also add a similarly named pair of optional methods for registering and deregistering VetoableChangeListeners:

```
void add<PropertyName>Listener(VetoableChangeListener p);
void remove<PropertyName>Listener(VetoableChangeListener p);
```

7.4.6 A bound and constrained example

In the following example the JellyBean class supports a bound property "color" and a bound and constrained property "priceInCents". We use instances of the PropertyChangeSupport and VetoableChangeSupport classes to help us keep track of the event listeners and to deliver the actual events.

```
import java.awt.*;
import java.beans.*;

public class JellyBean {

   public Color getColor() {
      return ourColor;
   }

   public void setColor(Color newColor) {
      Color oldColor = ourColor;
      ourColor = newColor;
      changes.firePropertyChange("color", oldColor, newColor);
   }

   public int getPriceInCents() {
      return ourPriceInCents;
   }

   public void setPriceInCents(int newPriceInCents)
                              throws PropertyVetoException {
      int oldPriceInCents = ourPriceInCents;
      // First tell the vetoers about the change. If anyone objects, we
      // let the PropertyVetoException propagate back to our caller.
      vetos.fireVetoableChange("priceInCents",
                     new Integer(oldPriceInCents),
                     new Integer(newPriceInCents));
      // No one vetoed, so go ahead and make the change.
      ourPriceInCents = newPriceInCents;
      changes.firePropertyChange("priceInCents",
                     new Integer(oldPriceInCents),
                     new Integer(newPriceInCents));
   }

   public void addPropertyChangeListener(PropertyChangeListener l) {
      changes.addPropertyChangeListener(l);
   }

   public void removePropertyChangeListener(PropertyChangeListener l) {
      changes.removePropertyChangeListener(l);
   }

   public void addVetoableChangeListener(VetoableChangeListener l) {
      vetos.addVetoableChangeListener(l);
   }

   public void removeVetoableChangeListener(VetoableChangeListener l) {
      vetos.removeVetoableChangeListener(l);
   }

   private PropertyChangeSupport changes =
                        new PropertyChangeSupport(this);
   private VetoableChangeSupport vetos = new VetoableChangeSupport(this);
   private Color ourColor = Color.orange;
   private int ourPriceInCents = 2;
}
```

Class PropertyChangeEvent

```
public class java.beans.PropertyChangeEvent
    extends java.util.EventObject
```

A "PropertyChange" event gets delivered whenever a bean changes a "bound" or "constrained" property. A PropertyChangeEvent object is sent as an argument to the PropertyChangeListener and VetoableChangeListener methods.

Normally PropertyChangeEvents are accompanied by the name and the old and new value of the changed property. If the new value is a builtin type (such as int or boolean) it must be wrapped as the corresponding java.lang.* Object type (such as Integer or Boolean).

Null values may be provided for the old and the new values if their true values are not known.

An event source may send a "null" name string to indicate that an arbitrary set of if its properties have changed. In this case the old and new values should also be null.

Constructors

- **PropertyChangeEvent**

```
public PropertyChangeEvent(Object source,
                           String propertyName,
                           Object oldValue,
                           Object newValue)
```

Parameters:

source
: The bean that fired the event.

propertyName
: The programmatic name of the property that was changed.

oldValue
: The old value of the property.

newValue
: The new value of the property.

Methods

- **getNewValue**

```
public Object getNewValue()
```

Returns:
The new value for the property, expressed as an Object. May be null if multiple properties have changed.

- **getOldValue**

```
public Object getOldValue()
```

Returns:
The old value for the property, expressed as an Object. May be null if multiple properties have changed.

- **getPropagationId**

```
public Object getPropagationId()
```

The "propagationId" field is reserved for future use. In Beans 1.0 the sole requirement is that if a listener catches a PropertyChangeEvent and then fires a PropertyChangeEvent of its own, then it should make sure that if propagates the propagationId field from its incoming event to its outgoing event.

Returns:
the propagationId object associated with a bound/constrained property update.

- **`getPropertyName`**

`public String getPropertyName()`

Returns:
The programmatic name of the property that was changed. May be null if multiple properties have changed.

- **`setPropagationId`**

`public void setPropagationId(Object propagationId)`

Parameters:
propagationId
The propagationId object for the event.

Interface PropertyChangeListener

```
public interface java.beans.PropertyChangeListener
    extends java.util.EventListener
```

A "PropertyChange" event gets fired whenever a bean changes a "bound" property. You can register a PropertyChangeListener with a source bean so as to be notified of any bound property updates.

Methods

- **`propertyChange`**

```
public abstract void
propertyChange(PropertyChangeEvent evt)
```

This method gets called when a bound property is changed.

Parameters:

evt

A PropertyChangeEvent object describing the event source and the property that has changed.

Class PropertyChangeSupport

```
public class java.beans.PropertyChangeSupport
    extends java.lang.Object
    implements java.io.Serializable
```

This is a utility class that can be used by beans that support bound properties. Your can either inherit from this class or you can use an instance of this class as a member field of your bean and delegate various work to it.

Constructors

- **PropertyChangeSupport**

  `public PropertyChangeSupport(Object sourceBean)`

Methods

- **addPropertyChangeListener**

  ```
  public synchronized void
  addPropertyChangeListener(PropertyChangeListener listener)
  ```

  Add a PropertyChangeListener to the listener list.

  Parameters:
  - listener
    - The PropertyChangeListener to be added

- **firePropertyChange**

  ```
  public void
  firePropertyChange(String propertyName,
                     Object oldValue, Object newValue)
  ```

  Report a bound property update to any registered listeners. No event is fired if old and new are equal and non-null.

  Parameters:
  - propertyName
    - The programmatic name of the property that was changed.
  - oldValue
    - The old value of the property.
  - newValue
    - The new value of the property.

- **removePropertyChangeListener**

  ```
  public synchronized void
  removePropertyChangeListener(PropertyChangeListener listener)
  ```

  Remove a PropertyChangeListener from the listener list.

  Parameters:
  - listener
    - The PropertyChangeListener to be removed

Class PropertyVetoException

```
public class java.beans.PropertyVetoException
    extends java.lang.Exception
```

A PropertyVetoException is thrown when a proposed change to a property represents an unacceptable value.

Constructors

- **PropertyVetoException**

```
public
PropertyVetoException(String mess,
                      PropertyChangeEvent evt)
```

Parameters:

mess
: Descriptive message

evt
: A PropertyChangeEvent describing the vetoed change.

Methods

- **getPropertyChangeEvent**

```
public PropertyChangeEvent getPropertyChangeEvent()
```

Interface VetoableChangeListener

```
public interface java.beans.VetoableChangeListener
    extends java.util.EventListener
```

A VetoableChange event gets fired whenever a bean changes a "constrained" property. You can register a VetoableChangeListener with a source bean so as to be notified of any constrained property updates.

Methods

- **vetoableChange**

```
public abstract void
vetoableChange(PropertyChangeEvent evt)
     throws PropertyVetoException
```

This method gets called when a constrained property is changed. The recipient may throw a PropertyVetoException if it wishes the property change to be rolled-back.

Parameters:

evt

A PropertyChangeEvent object describing the event source and the property that has changed.

Class VetoableChangeSupport

```
public class java.beans.VetoableChangeSupport
    extends java.lang.Object
    implements java.io.Serializable
```

This is a utility class that can be used by beans that support constrained properties. Your can either inherit from this class or you can use an instance of this class as a member field of your bean and delegate various work to it.

Constructors

- **VetoableChangeSupport**

```
public VetoableChangeSupport(Object sourceBean)
```

Methods

- **addVetoableChangeListener**

```
public synchronized void
addVetoableChangeListener(VetoableChangeListener listener)
```

Add a VetoableListener to the listener list.

Parameters:
listener
The VetoableChangeListener to be added

- **fireVetoableChange**

```
public void
fireVetoableChange(String propertyName,
                   Object oldValue, Object newValue)
     throws PropertyVetoException
```

Report a vetoable property update to any registered listeners. If anyone vetos the change, then fire a new event reverting everyone to the old value and then rethrow the PropertyVetoException.

No event is fired if old and new are equal and non-null.

Parameters:
propertyName
The programmatic name of the property that was changed.
oldValue
The old value of the property.
newValue
The new value of the property.

- **removeVetoableChangeListener**

```
public synchronized void
removeVetoableChangeListener(VetoableChangeListener listener)
```

Remove a VetoableListener from the listener list.

Parameters:

listener
    The VetoableListener to be removed

8 Introspection

8.1 Overview

At runtime and in the builder environment we need to be able to figure out which properties, events, and methods a Java Bean supports. We call this process *introspection*.

We want Java Beans developers to be able to work entirely in terms of Java. We therefore want to avoid using any separate specification language for defining the behaviour of a Java Bean. Rather we'd like all of its behaviour to be specifiable in Java.

A key goal of Java Beans is to make it very easy to write simple components and to provide default implementations for most common tasks. Therefore, we'd like to be able to introspect on simple beans without requiring that the beans developer do a whole bunch of extra work to support introspection. However, for more sophisticated components we also want to allow the component developers full and precise control over which properties, events, and methods are exposed.

We therefore provide a composite mechanism. By default we will use a low level *reflection* mechanism to study the methods supported by a target bean and then apply simple *design patterns* to deduce from those methods what properties, events, and public methods are supported. However, if a bean implementor chooses to provide a BeanInfo class describing their bean then this BeanInfo class will be used to programmatically discover the beans behaviour.

To allow application builders and other tools to analyze beans, we provide an Introspector class that understands the various design patterns and standard interfaces and provides a uniform way of introspecting on different beans.

For any given bean instance we expect its introspection information to be immutable and not to vary in normal use. However if a bean is updated with a new improved set of class files, then of course its signatures may change.

8.2 Overview of *Design Patterns*

By the term *design patterns* we mean conventional names and type signatures for sets of methods and/or interfaces that are used for standard purposes. For example the use of getFoo and setFoo methods to retrieve and set the values of a "foo" property.

These design patterns have two uses. First they are a useful documentation hint for human programmers. By quickly identifying particular methods as fitting standard design patterns, humans can more quickly assimilate and use new classes. Secondly, we can write tools and libraries that recognize design patterns and use them to analyze and understand components. In particular for Java Beans we use automatic identification of design patterns as a way for tools to identify properties, events, and exported methods.

However, within Java Beans the use of method and type names that match design patterns is entirely optional. If a programmer is prepared to explicitly specify their properties, methods, and events using the BeanInfo interface then they can call their methods and types whatever they like. However, these methods and types will still have to match the required type signatures, as this is essential to their operation.

Although use of the standard naming patterns is optional, we strongly recommend their use as standard naming conventions are an extremely valuable documentation technique.

8.3 Design Patterns for Properties

8.3.1 Simple properties

By default, we use design patterns to locate properties by looking for methods of the form:

```
public <PropertyType> get<PropertyName>();
public void set<PropertyName>(<PropertyType> a);
```

If we discover a matching pair of "get<*PropertyName*>" and "set<*PropertyName*>" methods that take and return the same type, then we regard these methods as defining a read-write property whose name will be "*<propertyName>*". We will use the "get<*PropertyName*>" method to get the property value and the "set<*PropertyName*>" method to set the property value. The pair of methods may be located either in the same class or one may be in a base class and the other may be in a derived class.

If we find only one of these methods, then we regard it as defining either a read-only or a write-only property called "*<propertyName>*"

By default we assume that properties are neither *bound* nor *constrained* (see Section 7).

So a simple read-write property "foo" might be represented by a pair of methods:

```
public Wombat getFoo();
public void setFoo(Wombat w);
```

8.3.2 Boolean properties

In addition, for boolean properties, we allow a getter method to match the pattern:

```
public boolean is<PropertyName>();
```

This "is<*PropertyName*>" method may be provided instead of a "get<*PropertyName*>" method, or it may be provided in addition to a "get<*PropertyName*>" method.

In either case, if the "is<*PropertyName*>" method is present for a boolean property then we will use the "is<*PropertyName*>" method to read the property value.

An example boolean property might be:

```
public boolean isMarsupial();
public void setMarsupial(boolean m);
```

8.3.3 Indexed properties

If we find a property whose type is an array "*<PropertyElement>*[]", then we also look for methods of the form:

```
public <PropertyElement> get<PropertyName>(int a);
public void set<PropertyName>(int a, <PropertyElement> b);
```

If we find either kind of pattern then we assume that "*<propertyName>*" is an indexed property and that these methods can be used to read and/or write an indexed value.

Thus an indexed property "foo" might be represented by four accessor methods:

```
public Bah[] getFoo();
public void setFoo(Bah a[]);
public Bah getFoo(int a);
public void setFoo(int a, Bah b);
```

8.4 Design Patterns for Events

By default, we use the following design pattern to determine which events a bean multicasts. We look for a pair of methods of the form:

```
public void add<EventListenerType>(<EventListenerType> a)
public void remove<EventListenerType>(<EventListenerType> a)
```

where both methods take the same "<*EventListenerType*>" type argument, where the "<*EventListenerType*>" type extends the "java.util.EventListener" interface, where the first method starts with "add", the second method starts with "remove", and where the "*<EventListenerType>*" type name ends with "Listener".

This design pattern assumes that the Java Bean is acting as a multicast event source for the events specified in the "*<EventListenerType>*" interface.

So for example:

```
public void addFredListener(FredListener t);
public void removeFredListener(FredListener t);
```

defines a multicast event source.

8.4.1 Unicast event sources

As a special case, when we locate an event source using the design pattern described above we check if the "add*<EventListenerType>*" method throws the java.util.TooManyListenersException. If so, we assume that the event source is unicast and can only tolerate a single event listener being registered at a time.

So for example:

```
public void addJackListener(JackListener t)
                        throws java.util.TooManyListenersException;
public void removeJackListener(JackListener t);
```

defines a unicast event source for the "JackListener" interface.

8.5 Design Patterns for Methods

By default, we assume that all public methods of a Java Bean should be exposed as external methods within the component environment for access by other components or by scripting languages.

By default, this includes any property accessor methods, and any event listener registry methods.

8.6 Explicit Specification using a BeanInfo class

A Java Bean can also explicitly specify which properties, events, and methods it supports by providing a class that implements the BeanInfo interface.

The BeanInfo interface provides methods for learning about the events, properties, methods, and global information about a target bean. As part of delivering a bean, an ISV may also deliver a matching BeanInfo class whose name is formed by adding "BeanInfo" to the bean's class name.

Java Beans uses purely programmatic APIs to obtain introspection information through BeanInfo. However note that individual component developers have the option of using arbitrary private bean description files behind the scenes and simply providing BeanInfo classes that know how to read these files.

A bean's BeanInfo class may chose to only specify part of a bean's behavior. It might for example return an EventSetDescriptor array from the getEventSetDescriptors method to provide definitive information on the bean's events while opting to return null from the getMethodDescriptors method, which indicates that the information on exposed methods should be discovered by examining the bean itself.

Thus to obtain a complete picture of a bean, application tools should always use the Introspector interface (see next section) which combines the information from a variety of potential sources to construct a definitive BeanInfo descriptor for a target bean.

8.7 Analyzing a Bean

We allow both explicit specification of a bean's exposed properties/methods/events and also implicit analysis using design patterns.

To simplify access to this information, and to make sure that all tools apply the same analysis rules, we provide a class java.beans.Introspector that should be used to analyze a bean class. It allows you to obtain a BeanInfo object that comprehensively describes a target bean class.

The Introspector class walks over the class/superclass chain of the target class. At each level it checks if there is a matching BeanInfo class which provides explicit information about the bean, and if so uses that explicit information. Otherwise it uses the low level reflection APIs to study the target class and uses design patterns to analyze its behaviour and then proceeds to continue the introspection with its baseclass. (See the Introspector class definition for a full description of the analysis rules.)

The reason for this multi-level analysis is to allow ISVs to deliver complex beans with explicitly specified behaviour where these beans are then subclassed and extended in minor ways by end customers. We want to be able to use automatic introspection to analyze the extensions added by the end customers and then add in the explicit information provided by the ISV who wrote the baseclass.

8.8 Capitalization of inferred names.

When we use design patterns to infer a property or event name, we need to decide what rules to follow for capitalizing the inferred name. If we extract the name from the middle of a normal mixedCase style Java name then the name will, by default, begin with a capital letter.

Java programmers are accustomed to having normal identifiers start with lower case letters. Vigorous reviewer input has convinced us that we should follow this same conventional rule for property and event names.

Thus when we extract a property or event name from the middle of an existing Java name, we normally convert the first character to lower case. However to support the occasional use of all upper-case names, we check if the first two characters of the name are both upper case and if so leave it alone. So for example,

"FooBah" becomes "fooBah"

"Z" becomes "z"

"URL" becomes "URL"

We provide a method Introspector.decapitalize which implements this conversion rule.

8.9 Security

The high-level Introspection APIs only provide information on "public" methods of target beans. The normal JDK security manager will allow even untrusted applets to access public fields and methods using the java.lang.reflect APIs. However, while access to private methods and fields may be available to "trusted" application builders, this access will normally be denied to untrusted applets.

Class BeanDescriptor

```
public class java.beans.BeanDescriptor
    extends java.beans.FeatureDescriptor
```

A BeanDescriptor provides global information about a "bean", including its Java class, its displayName, etc.

This is one of the kinds of descriptor returned by a BeanInfo object, which also returns descriptors for properties, method, and events.

Constructors

- **BeanDescriptor**

```
public BeanDescriptor(Class beanClass)
     throws IntrospectionException
```

Create a BeanDescriptor for a bean that doesn't have a customizer.

Parameters:

beanClass
: The Class object of the Java class that implements the bean. For example sun.beans.OurButton.class.

- **BeanDescriptor**

```
public BeanDescriptor(Class beanClass,
                      Class customizerClass)
```

Create a BeanDescriptor for a bean that has a customizer.

Parameters:

beanClass
: The Class object of the Java class that implements the bean. For example sun.beans.OurButton.class.

customizerClass
: The Class object of the Java class that implements the bean's Customizer. For example sun.beans.OurButtonCustomizer.class.

Methods

- **getBeanClass**

```
public Class getBeanClass()
```

Returns:

The Class object for the bean.

- **getCustomizerClass**

```
public Class getCustomizerClass()
```

Returns:

The Class object for the bean's customizer. This may be null if the bean doesn't have a customizer.

Interface BeanInfo

```
public interface java.beans.BeanInfo
```

A bean implementor who wishes to provide explicit information about their bean may provide a BeanInfo class that implements this BeanInfo interface and provides explicit information about the methods, properties, events, etc, of their bean.

A bean implementor doesn't need to provide a complete set of explicit information. You can pick and choose which information you want to provide and the rest will be obtained by automatic analysis using low-level reflection of the bean classes' methods and applying standard design patterns.

You get the opportunity to provide lots and lots of different information as part of the various XyZDescriptor classes. But don't panic, you only really need to provide the minimal core information required by the various constructors.

See also the SimpleBeanInfo class which provides a convenient "noop" base class for BeanInfo classes, which you can override for those specific places where you want to return explicit info.

To learn about all the behaviour of a bean see the Introspector class.

Variables

- **ICON_COLOR_16x16**

  `public final static int ICON_COLOR_16x16`

  Constant to indicate a 16 x 16 color icon.

- **ICON_COLOR_32x32**

  `public final static int ICON_COLOR_32x32`

  Constant to indicate a 32 x 32 color icon.

- **ICON_MONO_16x16**

  `public final static int ICON_MONO_16x16`

  Constant to indicate a 16 x 16 monochrome icon.

- **ICON_MONO_32x32**

  `public final static int ICON_MONO_32x32`

  Constant to indicate a 32 x 32 monochrome icon.

Methods

- **getAdditionalBeanInfo**

  `public abstract BeanInfo[] getAdditionalBeanInfo()`

  This method allows a BeanInfo object to return an arbitrary collection of other BeanInfo objects that provide additional information on the current bean.

  If there are conflicts or overlaps between the information provided by different BeanInfo objects, then the current BeanInfo takes precedence over the getAdditionalBeanInfo objects, and later elements in the array take precedence over earlier ones.

Returns:

an array of BeanInfo objects. May return null.

- **`getBeanDescriptor`**

```
public abstract BeanDescriptor getBeanDescriptor()
```

Returns:

A BeanDescriptor providing overall information about the bean, such as its displayName, its customizer, etc. May return null if the information should be obtained by automatic analysis.

- **`getDefaultEventIndex`**

```
public abstract int getDefaultEventIndex()
```

A bean may have a "default" event that is the event that will mostly commonly be used by human's when using the bean.

Returns:

Index of default event in the EventSetDescriptor array returned by getEventSetDescriptors.

Returns -1 if there is no default event.

- **`getDefaultPropertyIndex`**

```
public abstract int getDefaultPropertyIndex()
```

A bean may have a "default" property that is the property that will mostly commonly be initially chosen for update by human's who are customizing the bean.

Returns:

Index of default property in the PropertyDescriptor array returned by getPropertyDescriptors.

Returns -1 if there is no default property.

- **`getEventSetDescriptors`**

```
public abstract EventSetDescriptor[]
getEventSetDescriptors()
```

Returns:

An array of EventSetDescriptors describing the kinds of events fired by this bean. May return null if the information should be obtained by automatic analysis.

- **`getIcon`**

```
public abstract Image getIcon(int iconKind)
```

This method returns an image object that can be used to represent the bean in toolboxes, toolbars, etc. Icon images will typically be GIFs, but may in future include other formats.

Beans aren't required to provide icons and may return null from this method.

There are four possible flavors of icons (16x16 color, 32x32 color, 16x16 mono, 32x32 mono). If a bean choses to only support a single icon we recommend supporting 16x16 color.

We recommend that icons have a "transparent" background so they can be rendered onto an existing background.

Parameters:

iconKind

The kind of icon requested. This should be one of the constant values ICON_COLOR_16x16, ICON_COLOR_32x32, ICON_MONO_16x16, or ICON_MONO_32x32.

Returns:

An image object representing the requested icon. May return null if no suitable icon is available.

- **`getMethodDescriptors`**

```
public abstract MethodDescriptor[]
getMethodDescriptors()
```

Returns:

An array of MethodDescriptors describing the externally visible methods supported by this bean. May return null if the information should be obtained by automatic analysis.

- **`getPropertyDescriptors`**

```
public abstract PropertyDescriptor[]
getPropertyDescriptors()
```

Returns:

An array of PropertyDescriptors describing the editable properties supported by this bean. May return null if the information should be obtained by automatic analysis.

Class EventSetDescriptor

```
public class java.beans.EventSetDescriptor
    extends java.beans.FeatureDescriptor
```

An EventSetDescriptor describes a group of events that a given Java bean fires.

The given group of events are all delivered as method calls on a single event listener interface, and an event listener object can be registered via a call on a registration method supplied by the event source.

Constructors

- **EventSetDescriptor**

```
public EventSetDescriptor(Class sourceClass,
                          String eventSetName,
                          Class listenerType,
                          String listenerMethodName)
     throws IntrospectionException
```

This constructor creates an EventSetDescriptor assuming that you are following the most simple standard design pattern where a named event "fred" is (1) delivered as a call on the single method of interface FredListener, (2) has a single argument of type FredEvent, and (3) where the FredListener may be registered with a call on an addFredListener method of the source component and removed with a call on a removeFredListener method.

Parameters:

sourceClass
: The class firing the event.

eventSetName
: The programmatic name of the event. E.g. "fred". Note that this should normally start with a lower-case character.

listenerType
: The target interface that events will get delivered to.

listenerMethodName
: The method that will get called when the event gets delivered to its target listener interface.

- **EventSetDescriptor**

```
public
EventSetDescriptor(Class sourceClass,
                   String eventSetName,
                   Class listenerClass,
                   String listenerMethodNames[],
                   String addListenerMethodName,
                   String removeListenerMethodName)
     throws IntrospectionException
```

This constructor creates an EventSetDescriptor from scratch using string names.

Parameters:

sourceClass
: The class firing the event.

eventSetName
: The programmatic name of the event set. Note that this should normally start with a lower-case charac-

ter.

listenerType

The Class of the target interface that events will get delivered to.

listenerMethodNames

The names of the methods that will get called when the event gets delivered to its target listener interface.

addListenerMethodName

The name of the method on the event source that can be used to register an event listener object.

removeListenerMethodName

The name of the method on the event source that can be used to de-register an event listener object.

- **EventSetDescriptor**

```
public
EventSetDescriptor(String eventSetName,
                   Class listenerType,
                   Method listenerMethods[],
                   Method addListenerMethod,
                   Method removeListenerMethod)
     throws IntrospectionException
```

This constructor creates an EventSetDescriptor from scratch using java.lang.reflect.Method and java.lang.Class objects.

Parameters:

eventSetName

The programmatic name of the event set.

listenerType

The Class for the listener interface.

listenerMethods

An array of Method objects describing each of the event handling methods in the target listener.

addListenerMethod

The method on the event source that can be used to register an event listener object.

removeListenerMethod

The method on the event source that can be used to de-register an event listener object.

- **EventSetDescriptor**

```
public
EventSetDescriptor(String eventSetName,
                   Class listenerType,
                   MethodDescriptor listenerMethodDescriptors[],
                   Method addListenerMethod,
                   Method removeListenerMethod)
     throws IntrospectionException
```

This constructor creates an EventSetDescriptor from scratch using java.lang.reflect.MethodDescriptor and java.lang.Class objects.

Parameters:

eventSetName

The programmatic name of the event set.

listenerType

The Class for the listener interface.

listenerMethodDescriptors

An array of MethodDescriptor objects describing each of the event handling methods in the target listener.

addListenerMethod
The method on the event source that can be used to register an event listener object.
removeListenerMethod
The method on the event source that can be used to de-register an event listener object.

Methods

- **getAddListenerMethod**

  `public Method getAddListenerMethod()`

  Returns:
  The method used to register a listener at the event source.

- **getListenerMethodDescriptors**

  `public MethodDescriptor[] getListenerMethodDescriptors()`

  Returns:
  An array of MethodDescriptor objects for the target methods within the target listener interface that will get called when events are fired.

- **getListenerMethods**

  `public Method[] getListenerMethods()`

  Returns:
  An array of Method objects for the target methods within the target listener interface that will get called when events are fired.

- **getListenerType**

  `public Class getListenerType()`

  Returns:
  The Class object for the target interface that will get invoked when the event is fired.

- **getRemoveListenerMethod**

  `public Method getRemoveListenerMethod()`

  Returns:
  The method used to register a listener at the event source.

- **isInDefaultEventSet**

  `public boolean isInDefaultEventSet()`

  Report if an event set is in the "default set".

  Returns:
  True if the event set is in the "default set". Defaults to "true".

- **isUnicast**

  `public boolean isUnicast()`

  Normally event sources are multicast. However there are some exceptions that are strictly unicast.

  Returns:
  True if the event set is unicast. Defaults to "false".

- **`setInDefaultEventSet`**

```
public void
setInDefaultEventSet(boolean inDefaultEventSet)
```

Mark an event set as being in the "default" set (or not). By default this is true.

Parameters:

unicast
: True if the event set is unicast.

- **`setUnicast`**

```
public void setUnicast(boolean unicast)
```

Mark an event set as unicast (or not).

Parameters:

unicast
: True if the event set is unicast.

Class FeatureDescriptor

```
public class java.beans.FeatureDescriptor
    extends java.lang.Object
```

The FeatureDescriptor class is the common baseclass for PropertyDescriptor, EventSetDescriptor, and MethodDescriptor, etc.

It supports some common information that can be set and retrieved for any of the introspection descriptors.

In addition it provides an extension mechanism so that arbitrary attribute/value pairs can be associated with a design feature.

Constructors

- **`FeatureDescriptor`**

  `public FeatureDescriptor()`

Methods

- **`attributeNames`**

  `public Enumeration attributeNames()`

  Returns:

  An enumeration of the local-independent names of any attributes that have been registered with setValue.

- **`getDisplayName`**

  `public String getDisplayName()`

  Returns:

  The localized display name for the property/method/event. This defaults to the same as its programmatic name from getName.

- **`getName`**

  `public String getName()`

  Returns:

  The programmatic name of the property/method/event

- **`getShortDescription`**

  `public String getShortDescription()`

  Returns:

  A localized short description associated with this property/method/event. This defaults to be the display name.

- **`getValue`**

  `public Object getValue(String attributeName)`

  Retrieve a named attribute with this feature.

  Parameters:

attributeName
The locale-independent name of the attribute

Returns:
The value of the attribute. May be null if the attribute is unknown.

- **`isExpert`**

```
public boolean isExpert()
```

The "expert" flag is used to distinguish between those features that are intended for expert users from those that are intended for normal users.

Returns:
True if this feature is intended for use by experts only.

- **`isHidden`**

```
public boolean isHidden()
```

The "hidden" flag is used to identify features that are intended only for tool use, and which should not be exposed to humans.

Returns:
True if this feature should be hidden from human users.

- **`setDisplayName`**

```
public void setDisplayName(String displayName)
```

Parameters:
displayName
The localized display name for the property/method/event.

- **`setExpert`**

```
public void setExpert(boolean expert)
```

The "expert" flag is used to distinguish between features that are intended for expert users from those that are intended for normal users.

Parameters:
expert
True if this feature is intended for use by experts only.

- **`setHidden`**

```
public void setHidden(boolean hidden)
```

The "hidden" flag is used to identify features that are intended only for tool use, and which should not be exposed to humans.

Parameters:
hidden
True if this feature should be hidden from human users.

- **`setName`**

```
public void setName(String name)
```

Parameters:
name

The programmatic name of the property/method/event

- **`setShortDescription`**

```
public void setShortDescription(String text)
```

You can associate a short descriptive string with a feature. Normally these descriptive strings should be less than about 40 characters.

Parameters:

text
: A (localized) short description to be associated with this property/method/event.

- **`setValue`**

```
public void
setValue(String attributeName, Object value)
```

Associated a named attribute with this feature.

Parameters:

attributeName
: The locale-independent name of the attribute

value
: The value.

Class IndexedPropertyDescriptor

```
public class java.beans.IndexedPropertyDescriptor
    extends java.beans.PropertyDescriptor
```

An IndexedPropertyDescriptor describes a property that acts like an array and has an indexed read and/or indexed write method to access specific elements of the array.

An indexed property may also provide simple non-indexed read and write methods. If these are present, they read and write arrays of the type returned by the indexed read method.

Constructors

- **IndexedPropertyDescriptor**

```
public
IndexedPropertyDescriptor(String propertyName,
                          Class beanClass)
      throws IntrospectionException
```

This constructor constructs an IndexedPropertyDescriptor for a property that follows the standard Java conventions by having getFoo and setFoo accessor methods, for both indexed access and array access.

Thus if the argument name is "fred", it will assume that there is an indexed reader method "getFred", a non-indexed (array) reader method also called "getFred", an indexed writer method "setFred", and finally a non-indexed writer method "setFred".

Parameters:

propertyName
: The programmatic name of the property.

beanClass
: The Class object for the target bean.

- **IndexedPropertyDescriptor**

```
public
IndexedPropertyDescriptor(String propertyName,
                          Class beanClass,
                          String getterName,
                          String setterName,
                          String indexedGetterName,
                          String indexedSetterName)
      throws IntrospectionException
```

This constructor takes the name of a simple property, and method names for reading and writing the property, both indexed and non-indexed.

Parameters:

propertyName
: The programmatic name of the property.

beanClass
: The Class object for the target bean.

getterName
: The name of the method used for reading the property values as an array. May be null if the property is write-only or must be indexed.

setterName
The name of the method used for writing the property values as an array. May be null if the property is read-only or must be indexed.
indexedGetterName
The name of the method used for reading an indexed property value. May be null if the property is write-only.
indexedSetterName
The name of the method used for writing an indexed property value. May be null if the property is read-only.

- **IndexedPropertyDescriptor**

```
public
IndexedPropertyDescriptor(String propertyName,
                          Method getter,
                          Method setter,
                          Method indexedGetter,
                          Method indexedSetter)
     throws IntrospectionException
```

This constructor takes the name of a simple property, and Method objects for reading and writing the property.

Parameters:
propertyName
The programmatic name of the property.
getter
The method used for reading the property values as an array. May be null if the property is write-only or must be indexed.
setter
The method used for writing the property values as an array. May be null if the property is read-only or must be indexed.
indexedGetter
The method used for reading an indexed property value. May be null if the property is write-only.
indexedSetter
The method used for writing an indexed property value. May be null if the property is read-only.

Methods

- **getIndexedPropertyType**

```
public Class getIndexedPropertyType()
```

Returns:
The Java Class for the indexed properties type. Note that the Class may describe a primitive Java type such as "int".

This is the type that will be returned by the indexedReadMethod.

- **getIndexedReadMethod**

```
public Method getIndexedReadMethod()
```

Returns:
The method that should be used to read an indexed property value. May return null if the property isn't indexed or is write-only.

- **getIndexedWriteMethod**

```
public Method getIndexedWriteMethod()
```

Returns:

The method that should be used to write an indexed property value. May return null if the property isn't indexed or is read-only.

Class IntrospectionException

```
public class java.beans.IntrospectionException
    extends java.lang.Exception
```

Thrown when an exception happens during Introspection.

Typical causes include not being able to map a string class name to a Class object, not being able to resolve a string method name, or specifying a method name that has the wrong type signature for its intended use.

Constructors

- **IntrospectionException**

  `public IntrospectionException(String mess)`

  Parameters:

  mess

  Descriptive message

Class Introspector

```
public class java.beans.Introspector
    extends java.lang.Object
```

The Introspector class provides a standard way for tools to learn about the properties, events, and methods supported by a target Java Bean.

For each of those three kinds of information, the Introspector will separately analyze the bean's class and superclasses looking for either explicit or implicit information and use that information to build a BeanInfo object that comprehensively describes the target bean.

For each class "Foo", explicit information may be available if there exists a corresponding "FooBeanInfo" class that provides a non-null value when queried for the information. We first look for the BeanInfo class by taking the full package-qualified name of the target bean class and appending "BeanInfo" to form a new class name. If this fails, then we take the final classname component of this name, and look for that class in each of the packages specified in the BeanInfo package search path.

Thus for a class such as "sun.xyz.OurButton" we would first look for a BeanInfo class called "sun.xyz.OurButtonBeanInfo" and if that failed we'd look in each package in the BeanInfo search path for an OurButtonBeanInfo class. With the default search path, this would mean looking for "sun.beans.infos.OurButtonBeanInfo".

If a class provides explicit BeanInfo about itself then we add that to the BeanInfo information we obtained from analyzing any derived classes, but we regard the explicit information as being definitive for the current class and its base classes, and do not proceed any further up the superclass chain.

If we don't find explicit BeanInfo on a class, we use low-level reflection to study the methods of the class and apply standard design patterns to identify property accessors, event sources, or public methods. We then proceed to analyze the class's superclass and add in the information from it (and possibly on up the superclass chain).

Methods

- **decapitalize**

```
public static String decapitalize(String name)
```

Utility method to take a string and convert it to normal Java variable name capitalization. This normally means converting the first character from upper case to lower case, but in the (unusual) special case when there is more than one character and both the first and second characters are upper case, we leave it alone.

Thus "FooBah" becomes "fooBah" and "X" becomes "x", but "URL" stays as "URL".

Parameters:

name
: The string to be decapitalized.

Returns:

The decapitalized version of the string.

- **getBeanInfo**

```
public static BeanInfo getBeanInfo(Class beanClass)
     throws IntrospectionException
```

Introspect on a Java bean and learn about all its properties, exposed methods, and events.

Parameters:

beanClass
The bean class to be analyzed.

Returns:
A BeanInfo object describing the target bean.

- **`getBeanInfo`**

```
public static BeanInfo
getBeanInfo(Class beanClass, Class stopClass)
     throws IntrospectionException
```

Introspect on a Java bean and learn all about its properties, exposed methods, below a given "stop" point.

Parameters:
bean
The bean class to be analyzed.
stopClass
The baseclass at which to stop the analysis. Any methods/properties/events in the stopClass or in its baseclasses will be ignored in the analysis.

- **`getBeanInfoSearchPath`**

```
public static String[] getBeanInfoSearchPath()
```

Returns:
The array of package names that will be searched in order to find BeanInfo classes.

- **`setBeanInfoSearchPath`**

```
public static void
setBeanInfoSearchPath(String path[])
```

Change the list of package names that will be used for finding BeanInfo classes.

Parameters:
path
Array of package names.

Class MethodDescriptor

```
public class java.beans.MethodDescriptor
    extends java.beans.FeatureDescriptor
```

A MethodDescriptor describes a particular method that a Java Bean supports for external access from other components.

Constructors

- **MethodDescriptor**

  `public MethodDescriptor(Method method)`

  Parameters:

  method
  The low-level method information.

- **MethodDescriptor**

  ```
  public
  MethodDescriptor(Method method,
                   ParameterDescriptor parameterDescriptors[])
  ```

  Parameters:

  method
  The low-level method information.

  parameterDescriptors
  Descriptive information for each of the method's parameters.

Methods

- **getMethod**

  `public Method getMethod()`

  Returns:

  The low-level description of the method

- **getParameterDescriptors**

  ```
  public ParameterDescriptor[]
  getParameterDescriptors()
  ```

  Returns:

  The locale-independent names of the parameters. May return a null array if the parameter names aren't known.

Class ParameterDescriptor

```
public class java.beans.ParameterDescriptor
    extends java.beans.FeatureDescriptor
```

The ParameterDescriptor class allows bean implementors to provide additional information on each of their parameters, beyond the low level type information provided by the java.lang.reflect.Method class.

Currently all our state comes from the FeatureDescriptor base class.

Class PropertyDescriptor

```
public class java.beans.PropertyDescriptor
    extends java.beans.FeatureDescriptor
```

A PropertyDescriptor describes one property that a Java Bean exports via a pair of accessor methods.

Constructors

- **PropertyDescriptor**

```
public PropertyDescriptor(String propertyName,
                          Class beanClass)
     throws IntrospectionException
```

Constructs a PropertyDescriptor for a property that follows the standard Java convention by having getFoo and setFoo accessor methods. Thus if the argument name is "fred", it will assume that the reader method is "getFred" and the writer method is "setFred". Note that the property name should start with a lower case character, which will be capitalized in the method names.

Parameters:

propertyName
: The programmatic name of the property.

beanClass
: The Class object for the target bean. For example sun.beans.OurButton.class.

- **PropertyDescriptor**

```
public PropertyDescriptor(String propertyName,
                          Class beanClass,
                          String getterName,
                          String setterName)
     throws IntrospectionException
```

This constructor takes the name of a simple property, and method names for reading and writing the property.

Parameters:

propertyName
: The programmatic name of the property.

beanClass
: The Class object for the target bean. For example sun.beans.OurButton.class.

getterName
: The name of the method used for reading the property value. May be null if the property is write-only.

setterName
: The name of the method used for writing the property value. May be null if the property is read-only.

- **PropertyDescriptor**

```
public
PropertyDescriptor(String propertyName,
                   Method getter, Method setter)
     throws IntrospectionException
```

This constructor takes the name of a simple property, and Method objects for reading and writing the property.

Parameters:

propertyName
: The programmatic name of the property.

getter
: The method used for reading the property value. May be null if the property is write-only.

setter
: The method used for writing the property value. May be null if the property is read-only.

Methods

- **`getPropertyEditorClass`**

  `public Class getPropertyEditorClass()`

  Returns:

  Any explicit PropertyEditor Class that has been registered for this property. Normally this will return "null", indicating that no special editor has been registered, so the PropertyEditorManager should be used to locate a suitable PropertyEditor.

- **`getPropertyType`**

  `public Class getPropertyType()`

  Returns:

  The Java type info for the property. Note that the "Class" object may describe a built-in Java type such as "int".

  This is the type that will be returned by the ReadMethod.

- **`getReadMethod`**

  `public Method getReadMethod()`

  Returns:

  The method that should be used to read the property value. May return null if the property can't be read.

- **`getWriteMethod`**

  `public Method getWriteMethod()`

  Returns:

  The method that should be used to write the property value. May return null if the property can't be written.

- **`isBound`**

  `public boolean isBound()`

  Updates to "bound" properties will cause a "PropertyChange" event to get fired when the property is changed.

  Returns:

  True if this is a bound property.

- **`isConstrained`**

  `public boolean isConstrained()`

  Attempted updates to "Constrained" properties will cause a "VetoableChange" event to get fired when the property is changed.

  Returns:

  True if this is a constrained property.

- **`setBound`**

  `public void setBound(boolean bound)`

  Updates to "bound" properties will cause a "PropertyChange" event to get fired when the property is changed.

  Parameters:

  bound
  True if this is a bound property.

- **`setConstrained`**

  `public void setConstrained(boolean constrained)`

  Attempted updates to "Constrained" properties will cause a "VetoableChange" event to get fired when the property is changed.

  Parameters:

  constrained
  True if this is a constrained property.

- **`setPropertyEditorClass`**

  ```
  public void
  setPropertyEditorClass(Class propertyEditorClass)
  ```

  Normally PropertyEditors will be found using the PropertyEditorManager. However if for some reason you want to associate a particular PropertyEditor with a given property, then you can do it with this method.

  Parameters:

  propertyEditorClass
  The Class for the desired PropertyEditor.

Class SimpleBeanInfo

```
public class java.beans.SimpleBeanInfo
    extends java.lang.Object
    implements java.beans.BeanInfo
```

This is a support class to make it easier for people to provide BeanInfo classes.

It defaults to providing "noop" information, and can be selectively overriden to provide more explicit information on chosen topics. When the introspector sees the "noop" values, it will apply low level introspection and design patterns to automatically analyze the target bean.

Methods

- **getAdditionalBeanInfo**

  `public BeanInfo[] getAdditionalBeanInfo()`

  Claim there are no other relevant BeanInfo objects. You may override this if you want to (for example) return a BeanInfo for a base class.

- **getBeanDescriptor**

  `public BeanDescriptor getBeanDescriptor()`

  Deny knowledge about the class and customizer of the bean. You can override this if you wish to provide explicit info.

- **getDefaultEventIndex**

  `public int getDefaultEventIndex()`

  Deny knowledge of a default event. You can override this if you wish to define a default event for the bean.

- **getDefaultPropertyIndex**

  `public int getDefaultPropertyIndex()`

  Deny knowledge of a default property. You can override this if you wish to define a default property for the bean.

- **getEventSetDescriptors**

  `public EventSetDescriptor[] getEventSetDescriptors()`

  Deny knowledge of event sets. You can override this if you wish to provide explicit event set info.

- **getIcon**

  `public Image getIcon(int iconKind)`

  Claim there are no icons available. You can override this if you want to provide icons for your bean.

- **getMethodDescriptors**

  `public MethodDescriptor[] getMethodDescriptors()`

  Deny knowledge of methods. You can override this if you wish to provide explicit method info.

- **getPropertyDescriptors**

```
public PropertyDescriptor[] getPropertyDescriptors()
```

Deny knowledge of properties. You can override this if you wish to provide explicit property info.

- **`loadImage`**

```
public Image loadImage(String resourceName)
```

This is a utility method to help in loading icon images. It takes the name of a resource file associated with the current object's class-loader and loads an image object from that file. Typically images will be GIFs.

Parameters:

resourceName
: A pathname relative to the base directory of the current class-loader. For example "sun/beans/wombat.gif".

Returns:

an image object. May be null if the load failed.

9 Customization

When a user is composing an application in an application builder we want to allow them to customize the appearance and behaviour of the various beans they are using.

We allow this customization to occur in two different ways. When a bean exports a set of properties, then an application builder can use these properties to construct a GUI *property sheet* that lists the properties and provides a *property editor* for each property. The user can then use this property sheet to update the various properties of the bean.

This kind of simple property sheet is likely to work well for simple to medium sized beans. However for larger and more complex beans, we want to allow more sophisticated kinds of component customization. For example, we would like to allow component writers to provide customization "wizards" that guide users through the different steps of customizing a bean, rather than simply facing them with property sheet choices.

We therefore allow each Java Bean to be accompanied by a *customizer* class that controls the customization of the bean. This customizer class should be an AWT component that can be run to customize a target bean. It can provide whatever GUI behaviour it wishes to control the customization of the target bean.

9.1 Storing customized components

When a human has customized the behaviour of a set of components, the application builder should use the persistence mechanism described in Section 5 to store away the state of the components. When the application is run, this pickled state should be read back in to initialize the components.

9.2 Property editors

A *property editor* is a class that exists to allow GUI editing of a property value such as a String or Color. Bean developers may also deliver new property editors for any new data types that they deliver as part of their bean. For example, if a database bean developer defines a new type foo.baz.SQLString and uses that type for component properties, then they may also wish to provide a property editor foo.baz.SQLStringEditor for editing SQLString values.

The PropertyEditor class provides a number of different ways of reading and writing the property value. Not all property editors need support all the different options. At the low-end a PropertyEditor may simply support reading and writing a value as a String. At the high-end, a PropertyEditor is allowed to provide a full-blown java.awt.Component that can be popped-up to edit its value.

A property editor may be instantiated either as part of a property sheet, or as a field within a more complex component customizer. If a PropertyEditor allows itself to be represented in several different ways (as a String, as a painted rectangle, as a full-blown custom editor component, etc.) is the responsibility of the higher-level tool to decide how best to represent a given PropertyEditor.

9.2.1 Locating property editors

Some property editors will be supplied as part of the Java Beans runtimes, other property editors may be supplied by component developers, other property editors may be provided by application builder tools.

To allow the right property editor to be located for a given Java data type, we provide a class java.beans.PropertyEditorManager that maps between Java types and corresponding property editor classes.

The EditorManager class manages a registry of property editors, and allows you to register a class to act as the property editor for a given Java type. The registry is pre-loaded with editors for the standard Java built-in types such as "int", "boolean", etc.

When you attempt to locate a property editor for a Java type, the property manager searches for a suitable property editor by:

- First looking to see if a property editor has been explicitly registered.
- If that fails, it will take the given Java type name and add "Editor" to the end and look for that class. So for "foo.bah.Wombat" it will look for "foo.bah.WombatEditor" to act as the property editor.
- If that also fails, it will take the final component of the given class name, add "Editor" to the end and then look for that class name in a search-list of packages. (By default java.beans.editors). Thus for "foo.bah.Wombat" it will check for "java.beans.editors.WombatEditor". Application builder tools may change the package search path, by for example, pre-pending a package that they themselves provide.

Normally component writers who define new types and want to supply property editors should simply provide an editor class whose name is the base type name + "Editor".

Application builder tools can either use the set of default property editors provided with Java Beans or define their own set of property editors for the basic Java types.

9.2.2 Reporting changes

Whenever a property editor makes a change it should fire a "PropertyChange" event. This allows higher level software to detect the change, retrieve the new value, and do an appropriate setXXX call to the target component.

9.2.3 Don't change the initial object

Property editors are given an object representing the current value of a property. However a property editor should not directly change this initial object. Instead, based on user input, it should create a new object to reflect the revised state of the property. When the property editor fires a "PropertyChange" event (see above) higher level software will detect the change, retrieve the new object, and do an appropriate property setter method call on the target component to set the new value.

Since most simple objects in Java (String, Color, Font) are immutable, this model works naturally for them. However, for mutable data types this also feels like the clearest and most reliable model. It seems likely to lead to errors and misunderstandings if we were to try to change an existing property object's value under a component's feet.

Some property sheets may immediately apply each modified property value to the target component. However some property sheets (or customizers) may chose to delay the changes until the users confirms them.

9.2.4 Indirect property changes

In some components, setting one property value may cause other property values to change.

If this happens while a property sheet is open, then the user would like the property sheet to be updated with all the new values. We therefore recommend that after each property update a property sheet manager should re-read all the properties associated with the target bean. However, in order to avoid excessive screen redraws we recommend that it should do an "==" followed by a "Object.equals" test to compare each new property value with its previous value. The individual property editors should only be updated if the value has actually changed.

We considered defining an "IndirectPropertyUpdate" event that should get fired when a property was changed as a side-effect of some other operation. However this seemed to add extra weight to the interface for only a fairly small performance gain.

See also "bound properties" in Section 7.4.

9.3 Customizers

Property sheets will be an adequate way of customizing most simple components.

However for large components that offer a number of different ways of being configured, it may be more appropriate for the component developer to provide a special class for customizing the component. We refer to these classes as *customizers*.

Each customizer should inherit either directly or indirectly from java.awt.Component. It should also implement the java.beans.Customizer interface.

9.3.1 GUI appearance

Normally each Customizer will be run in a separate AWT dialog window. The customizer has complete discretion how it chooses to represent itself, and may redraw its appearance as the user navigates moves through different stages of customization.

A customizer may chose to include property editors in its screen appearance to edit target data values.

9.3.2 In place update

Each customizer will be given a target object to customize as its constructor argument.

It is entirely up to the individual customizer whether it chooses to apply changes as it goes along, or whether it batches up changes and applies them to the object at appropriate points. However, whenever possible, we advise that the target bean's screen appearance should be quickly updated to reflect requested changes.

9.3.3 Locating customizers

If a bean wishes to provide its own Customizer then it must provide its own BeanInfo class (see 8.6). Then tools can use the beanInfo.getCustomizerClass() method to locate the bean's Customizer class.

Interface Customizer

```
public interface java.beans.Customizer
```

A customizer class provides a complete custom GUI for customizing a target Java Bean.

Each customizer should inherit from the java.awt.Component class so it can be instantiated inside an AWT dialog or panel.

Each customizer should have a null constructor.

Methods

- **addPropertyChangeListener**

  ```
  public abstract void
  addPropertyChangeListener(PropertyChangeListener listener)
  ```

  Register a listener for the PropertyChange event. The customizer should fire a PropertyChange event whenever it changes the target bean in a way that might require the displayed properties to be refreshed.

  Parameters:

  listener
  : An object to be invoked when a PropertyChange event is fired.

- **removePropertyChangeListener**

  ```
  public abstract void
  removePropertyChangeListener(PropertyChangeListener listener)
  ```

  Remove a listener for the PropertyChange event.

  Parameters:

  listener
  : The PropertyChange listener to be removed.

- **setObject**

  ```
  public abstract void setObject(Object bean)
  ```

  Set the object to be customized. This method should be called only once, before the Customizer has been added to any parent AWT container.

  Parameters:

  bean
  : The object to be customized.

Interface PropertyEditor

```
public interface java.beans.PropertyEditor
```

A PropertyEditor class provides support for GUIs that want to allow users to edit a property value of a given type.

PropertyEditor supports a variety of different kinds of ways of displaying and updating property values. Most PropertyEditors will only need to support a subset of the different options available in this API.

Simple PropertyEditors may only support the getAsText and setAsText methods and need not support (say) paintValue or getCustomEditor. More complex types may be unable to support getAsText and setAsText but will instead support paintValue and getCustomEditor.

Every propertyEditor must support one or more of the three simple display styles. Thus it can either (1) support isPaintable or (2) return a non-null String[] from getTags() or (3) return a non-null String from getAsText().

Every property editor must support a call on setValue when the argument object is of the type for which this is the corresponding propertyEditor. In addition, each property editor must either support a custom editor, or support setAsText.

Each PropertyEditor should have a null constructor.

Methods

- **addPropertyChangeListener**

```
public abstract void
addPropertyChangeListener(PropertyChangeListener listener)
```

Register a listener for the PropertyChange event. When a PropertyEditor changes its value it should fire a PropertyChange event on all registered PropertyChangeListeners, specifying a null string for the property name and itself as the source.

Parameters:

listener

An object to be invoked when a PropertyChange event is fired.

- **getAsText**

```
public abstract String getAsText()
```

Returns:

The property value as a human editable string.

Returns null if the value can't be expressed as an editable string.

If a non-null value is returned, then the PropertyEditor should be prepared to parse that string back in setAsText().

- **getCustomEditor**

```
public abstract Component getCustomEditor()
```

A PropertyEditor may choose to make available a full custom Component that edits its property value. It is the responsibility of the PropertyEditor to hook itself up to its editor Component itself and to report property value changes by firing a PropertyChange event.

The higher-level code that calls getCustomEditor may either embed the Component in some larger property sheet, or it may put it in its own individual dialog, or ...

Returns:

A java.awt.Component that will allow a human to directly edit the current property value. May be null if this is not supported.

- **`getJavaInitializationString`**

```
public abstract String getJavaInitializationString()
```

This method is intended for use when generating Java code to set the value of the property. It should return a fragment of Java code that can be used to initialize a variable with the current property value.

Example results are "2", "new Color(127,127,34)", "Color.orange", etc.

Returns:

A fragment of Java code representing an initializer for the current value.

- **`getTags`**

```
public abstract String[] getTags()
```

If the property value must be one of a set of known tagged values, then this method should return an array of the tags. This can be used to represent (for example) enum values. If a PropertyEditor supports tags, then it should support the use of setAsText with a tag value as a way of setting the value.

Returns:

The tag values for this property. May be null if this property cannot be represented as a tagged value.

- **`getValue`**

```
public abstract Object getValue()
```

Returns:

The value of the property. Builtin types such as "int" will be wrapped as the corresponding object type such as "java.lang.Integer".

- **`isPaintable`**

```
public abstract boolean isPaintable()
```

Returns:

True if the class will honor the paintValue method.

- **`paintValue`**

```
public abstract void
paintValue(Graphics gfx, Rectangle box)
```

Paint a representation of the value into a given area of screen real estate. Note that the propertyEditor is responsible for doing its own clipping so that it fits into the given rectangle.

If the PropertyEditor doesn't honor paint requests (see isPaintable) this method should be a silent noop.

Parameters:

gfx

Graphics object to paint into.

box

Rectangle within graphics object into which we should paint.

- **`removePropertyChangeListener`**

```
public abstract void
removePropertyChangeListener(PropertyChangeListener listener)
```

Remove a listener for the PropertyChange event.

Parameters:
listener
The PropertyChange listener to be removed.

- **`setAsText`**

```
public abstract void setAsText(String text)
      throws IllegalArgumentException
```

Set the property value by parsing a given String. May raise java.lang.IllegalArgumentException if either the String is badly formatted or if this kind of property can't be expressed as text.

Parameters:
text
The string to be parsed.

- **`setValue`**

```
public abstract void setValue(Object value)
```

Set (or change) the object that is to be edited. Builtin types such as "int" must be wrapped as the corresponding object type such as "java.lang.Integer".

Parameters:
value
The new target object to be edited. Note that this object should not be modified by the PropertyEditor, rather the PropertyEditor should create a new object to hold any modified value.

- **`supportsCustomEditor`**

```
public abstract boolean supportsCustomEditor()
```

Returns:
True if the propertyEditor can provide a custom editor.

Class PropertyEditorManager

```
public class java.beans.PropertyEditorManager
    extends java.lang.Object
```

The PropertyEditorManager can be used to locate a property editor for any given type name. This property editor must support the java.beans.PropertyEditor interface for editing a given object.

The PropertyEditorManager uses three techniques for locating an editor for a given type. First, it provides a registerEditor method to allow an editor to be specifically registered for a given type. Second it tries to locate a suitable class by adding "Editor" to the full qualified classname of the given type (e.g. "foo.bah.FozEditor"). Finally it takes the simple classname (without the package name) adds "Editor" to it and looks in a search-path of packages for a matching class.

So for an input class foo.bah.Fred, the PropertyEditorManager would first look in its tables to see if an editor had been registered for foo.bah.Fred and if so use that. Then it will look for a foo.bah.FredEditor class. Then it will look for (say) standardEditorsPackage.FredEditor class.

Methods

- **findEditor**

  ```
  public static PropertyEditor
  findEditor(Class targetType)
  ```

  Locate a value editor for a given target type.

  Parameters:

  targetType
  : The Class object for the type to be edited

  Returns:
  : An editor object for the given target class. The result is null if no suitable editor can be found.

- **getEditorSearchPath**

  ```
  public static String[] getEditorSearchPath()
  ```

  Returns:
  : The array of package names that will be searched in order to find property editors.

- **registerEditor**

  ```
  public static void
  registerEditor(Class targetType, Class editorClass)
  ```

  Register an editor class to be used to editor values of a given target class.

  Parameters:

  targetType
  : the Class object of the type to be edited

  editorClass
  : the Class object of the editor class

- **setEditorSearchPath**

```
public static void
setEditorSearchPath(String path[])
```

Change the list of package names that will be used for finding property editors.

Parameters:

path

Array of package names.

10 Miscellaneous

10.1 java.beans.Beans

The java.beans.Beans class provides a few miscellaneous beans control methods.

The isDesignTime and isGuiAvailable methods allow a bean to test its environment so as to tailor its behaviour.

10.2 java.beans.Visibility

Some beans may be run in both client GUI application and in "invisible" server applications. The java.beans.Visibility interface lets a container instruct a bean if it is in a server only environment.

This interface need only be supported by those beans that want to provide different behaviour in a server environment.

10.3 Instantiating a bean

A bean must be instantiated in one of two different ways, depending on whether it provides a serialized prototype object or not.

Non-serialized beans may be instantiated by simply using "new" as for normal classes:

```
JellyBean fred = new JellyBean();
```

If a bean provides a serialized template, then it must be instantiated by opening the resource file associated with the serialized state and reading in the serialized object. To make this more convenient we provide a utility method java.beans.Beans.instantiate that can read in a serialized object from a named resource file at a target class loader:

```
ClassLoader cl = Class.forName("acme.widgets.Foo").getClassLoader();
MutantWombat w = (MutantWombat) Beans.instantiate(cl,
                              "/acme/widgets/PurpleMutantWombat.ser");
```

10.4 Obtaining different type views of a Java Bean

In Beans 1.0 each bean is represented by a single Java object. However in future versions of Java Beans we may wish to add support for more complex beans, where a set of Java objects cooperate to provide a set of differently typed views that are all part of one bean.

Therefore when an application program wishes to determine if a Java Bean supports a target interface or if it wishes to cast a Java Bean to a target interface, we strongly recommend that the Beans.instanceOf and Beans.getInstanceOf methods are used. These methods are initially implemented in Beans 1.0 simply in terms of the Java "instanceof" operator, but as we evolve the Java Beans APIs this may change. So developers are encouraged to use these methods now to allow their application code to work with future extensions.

So a typical sequence for attempting to assess if a bean supports a particular API might be:

```
sunw.DataseBaseAccessor x = getDBAccessor();
java.awt.Component y = null;
Class targetClass = Class.forName("java.awt.Component");
if (Beans.isInstanceOf(x, targetClass)) {
    y = (java.awt.Component) Beans.getInstanceOf(x, targetClass);
}
```

Class Beans

```
public class java.beans.Beans
    extends java.lang.Object
```

This class provides some general purpose beans control methods.

Methods

- **getInstanceOf**

```
public static Object
getInstanceOf(Object bean, Class targetType)
```

From a given bean, obtain an object representing a specified type view of that source object.

The result may be the same object or a different object. If the requested target view isn't available then the given bean is returned.

This method is provided in Beans 1.0 as a hook to allow the addition of more flexible bean behaviour in the future.

Parameters:

obj
: Object from which we want to obtain a view.

targetType
: The type of view we'd like to get.

- **instantiate**

```
public static Object
instantiate(ClassLoader cls, String name)
     throws IOException, ClassNotFoundException
```

Read in a bean from a serialized object. The serialized object is a named resource file available from a given class-loader.

Parameters:

classLoader
: The class-loader from which we should read the serialized object. If this is null then the system class-loader is used.

name
: This names the serialized object within the class-loader. For example "sun/beanbox/foobah.ser"

- **isDesignTime**

```
public static boolean isDesignTime()
```

Returns:

True if we are running in an application construction environment.

- **isGuiAvailable**

```
public static boolean isGuiAvailable()
```

Returns:

True if we are running in an environment where beans can assume that an interactive GUI is available, so they can pop up dialog boxes, etc. This will normally return true in a windowing environment, and will normally return false in a server environment or if an application is running as part of a batch job.

- **isInstanceOf**

```
public static boolean
isInstanceOf(Object bean, Class targetType)
```

Check if a bean can be viewed as a given target type. The result will be true if the Beans.getInstanceof method can be used on the given bean to obtain an object that represents the specified targetType type view.

Parameters:

bean
: Bean from which we want to obtain a view.

targetType
: The type of view we'd like to get.

Returns:

"true" if the given bean supports the given targetType.

- **setDesignTime**

```
public static void
setDesignTime(boolean isDesignTime)
     throws SecurityException
```

Used to indicate whether of not we are running in an application builder environment. Note that this method is security checked and is not available to (for example) untrusted applets.

Parameters:

isDesignTime
: True if we're in an application builder tool.

- **setGuiAvailable**

```
public static void
setGuiAvailable(boolean isGuiAvailable)
     throws SecurityException
```

Used to indicate whether of not we are running in an environment where GUI interaction is available. Note that this method is security checked and is not available to (for example) untrusted applets.

Parameters:

isGuiAvailable
: True if GUI interaction is available.

Interface Visibility

```
public interface java.beans.Visibility
```

Under some circumstances a bean may be run on servers where a GUI is not available. This interface can be used to query a bean to determine whether it absolutely needs a gui, and to advise the bean whether a GUI is available.

This interface is for expert developers, and is not needed for normal simple beans. To avoid confusing end-users we avoid using getXXX setXXX design patterns for these methods.

Methods

- **avoidingGui**

```
public abstract boolean avoidingGui()
```

Returns:
true if the bean is currently avoiding use of the Gui. e.g. due to a call on dontUseGui().

- **dontUseGui**

```
public abstract void dontUseGui()
```

This method instructs the bean that it should not use the Gui.

- **needsGui**

```
public abstract boolean needsGui()
```

Returns:
True if the bean absolutely needs a GUI available in order to get its work done.

- **okToUseGui**

```
public abstract void okToUseGui()
```

This method instructs the bean that it is OK to use the Gui.

11 Packaging

11.1 Goals and Non Goals

This chapter specifies the format and conventions needed so that a packaged Java Bean can be used by tools and applications.

The Java Beans specification does *not* prescribe a format to be used by an application to store the Java Beans it uses. In particular, for the special but very important case of an application builder, the Java Beans specification does *not* prescribe a format for projects, *nor* does it prescribe a format for the delivery of built applications. So for example, although beans are initially delivered as JAR files, they may be arbitrarily repackaged when they are built into an application.

11.2 Overview

Packaging of Java Beans is based on two technologies available in JDK1.1: Java object serialization and JAR files. Object serialization is used to store the state of Java objects, in particular Java Beans. JAR files are used to collect class files, serialized objects, images, help files and similar *resource* files

A JAR file is a ZIP format archive file that may optionally have a *manifest file* with additional information describing the contents of the JAR file (see Section 11.3.3 below). A single JAR file may contain more than one Java Bean; this simplifies packaging and allows for sharing of classes and resource files at packaging time.

Classes in a JAR file may be signed; this may grant additional privileges to them (class signing uses information that is also stored in the manifest file).

The beans-related operations to perform on a JAR file are:

- Given a bean name, extract the information for that bean from a JAR file,
- List all the beans available in a given JAR file.

11.3 Content

A Java Bean JAR file contains a number of entries defining one or more packaged Java Beans. By convention, JAR file entries are given slash-separated names such as “a/b/c”.

Each Java Bean may include entries defining:

1. A set of classes to provide behavior for the Java Bean. These entries must have names ending in “.class”. E.g. “foo/bah/Aardvark.class”.
2. A serialized prototype of a bean to be used to initialize the bean. These entries must have names ending in “.ser”. E.g. “foo/elk.ser”.

   One of 1 or 2 must be present; see below.
3. Optional help files in HTML format to provide documentation for the Java Bean. If the Java Bean is localized, there might be multiple versions of this documentation, one per locale. These entries must have names of the form *<locale>/.../<tail>.html*.

4. Optional internationalization information to be used by the bean to localize itself.
5. Other *resource* files needed by the Java Bean (like images, sound, video, whatever). This is open-ended.

11.3.1 Names of Java Beans

The name of a bean is a String comprising a "." separated sequence of names. Bean names are unique in their domain (which may be the CLASSPATH, or the domain of a classloader). The name of a bean is the name of its class, or of its serialized form.

To avoid naming collisions, bean developers should follow the standard Java de facto naming convention for classes, where the top-level package name is a company name.

11.3.2 Identifying Java Beans in a JAR file

Each class in a JAR package is potentially a bean; each serialized object in a JAR package is potentially a bean.

If an entry in the JAR file has name "*a/b/c.class*", then the name of its associated bean is "*a.b.c*". Similarly, if an entry in the JAR file has name "*x/y/z.ser*", then the name of its associated bean is "*x.y.z*".

A JAR file that wants to advertise its contents should have a manifest file. The manifest file will indicate which of the potential beans in the JAR archive are actual beans. If there is no manifest file every potential Bean is regarded as one.

JDK1.1 provides an API to navigate the contents of a JAR file.

11.3.3 Format of Manifest File

As part of the JAR file format, a JAR archive may include a *manifest file* to describe its contents. (See the Manifest File specification for a full description.)

The manifest file must be named "META-INF/MANIFEST.MF".

The manifest file in a JAR archive provides information on selected parts of the contents of the archive. It is a sequence of *sections* separated by empty lines. Each section contains one or more *headers*, (so named because they conform to the RFC822 standard) each of the form *<name>: <value>*. The sections that provide information on files in the archive must have a header whose *<name>* is **Name**, and whose *<value>* is the relative name of the file being described.

A manifest file can include information describing what beans are in the JAR archive. If a file in the JAR archive is a Java Bean, its corresponding section should contain a header whose *<name>* is **Java-Bean**, and whose *<value>* is **True** (case is not important).

For example, two relevant sections of a manifest file might be:

```
Name: wessex/wonder/bean.class
Java-Bean: True

Name: wessex/quaint/bean.ser
Java-Bean: True
Rustic: True
Victorian: True
```

11.3.4 Accessing Resource Files

Java code occasionally needs to access some information stored in a file. Some examples are images, audio files, text messages, as well as serialized objects.

JDK1.1 provides a basic mechanism to specify a *resource* used by a class in a way that is independent of where the class actually resides (e.g. in a CLASSPATH, or through a ClassLoader). Access to persistent representations of Beans (Section 11.3.5) uses this basic mechanism, as do the I18N facilities in JDK1.1.

A typical use could be:

```
Class c = Class.forName("foo.bah.XyzComponent");
InputStream x = c.getResourceAsStream("/foo/bah/MyRedStuff.txt");
```

See the section on *resources* and *Internationalization* in JDK1.1 for more details.

11.3.5 Persistence representation

A JAR file may contain serialized prototypes of Java Beans. Not all Java Beans need to have a prototype. Those that do will use the Java Object Serialization mechanism (see Chapter 5).

The prototypes are stored and restored by a simple use of the *java.io.ObjectInputStream* and *java.io.ObjectOutputStream*. Thus, to serialize

```
java.io.OutputStream out = ... // the stream where to serialize
java.io.ObjectOutputStream s = new java.io.ObjectOutputStream(out);
s.writeObject(myComponent);
```

Deserialization is a little more complex. We will normally want to be able to deserialize a serialized object from any arbitrary class-loader. So we have to first locate a class that we can use to identify the source class-loader, and then use Beans.instantiate to read in a serialized object from a named file at that class-loader.

```
ClassLoader cl = Class.forName("foo.bah.Xyz").getClassLoader();
MyStuff x = (MyStuff) Beans.instantiate(cl,"/foo/bah/MyRedStuff.ser");
```

When an object is deserialized from a stream we will use that same class-loader as a source for .class files to satisfy any class dependencies.

See the documentation on Serialization for more details.

11.3.6 Help Documentation.

Documentation is described as a collection of HTML entries. These entries should be in HTML 2.0 format. An application using Java Beans may choose to translate the contents of these entries into whatever format it chooses before presenting it to the user.

Java Beans writers are strongly encouraged to include help entries for their beans.

For each bean the optional top-level help documentation can in *<locale>*/*<bean-name>.html*. (If the documentation is only available for a single locale, the initial *<locale>*/ may be omitted.)

Cross references to other information can be presented as hyperlinks, either as relative URLs to other html files in the same JAR file, or as absolute URLs that lead back to the vendor's web site.

Appendix A: Transitional Beans under JDK 1.0.2

A.1 Goals.

Some of the features of the full 1.0 Java Beans architecture rely on new Java core interfaces such as reflection and serialization that are being added in the JDK 1.1.

However in order to allow the early development and deployment of beans we want to allow developers to produce beans that can run under JDK 1.0.2, but which are also fully compatible with the final beans architecture.

To enable this, we are defining a set of rules for “transitional” Beans, which are fully compliant with the 1.0 Java Beans architecture, but which can be used to build applets and applications that can run on JDK 1.0.2 and on JDK 1.0.2 enabled systems such as Netscape Navigator 3.0 and Internet Explorer 3.0.

These transitional beans will necessarily only use a subset of the full bean model. However this subset includes the basic property/method/event mechanisms, so that a transitional bean is still very useful within a JDK 1.0.2 environment.

These transitional beans are primarily intended for use with leading edge application builder tools.

A.2 Marker classes

In several places the Java Beans APIs use particular classes or interfaces as “markers” to determine that a given class is intended to perform a given role. For example, an event listener class must implement the java.util.EventListener interface.

We would like transitional Beans to use markers for events and serialization, so that these markers can be recognized correctly when the transitional beans are used in full JDK 1.1 environments. We require that transitional beans should use three transitional marker classes (sunw.util.EventObject, sunw.util.EventListener, sunw.io.Serializable) as needed as markers.

The fine print explains why.

We could simply make available the appropriate JDK 1.1 marker classes to developers for use with JDK 1.0.2. However these marker classes are all in the java.* part of the namespace and many existing browsers are reluctant to allow downloading of java.* classes.

To solve this problem, JavaSoft is making available three interim marker classes (sunw.util.EventObject, sunw.util.EventListener and sunw.io.Serializable) that can be downloaded over the net into JDK 1.0.2 browsers. These classes will be mostly noops under 1.0.2, but as part of JDK 1.1 JavaSoft will distribute replacement versions of these classes that correctly inherit from the real marker classes.

Thus a transitional bean might inherit from sun.io.Serializable to indicate that it is compatible with the JDK 1.1 serialization mechanism. When this bean is downloaded into a JDK 1.0.2 system then the JDK 1.0.2 version of sunw.io.Serializable (which is basically an empty interface) can be downloaded as part of the bean. This will allow the bean to run happily on JDK 1.0.2. However when the same bean is run on a JDK 1.1 system then the class loader will use the JDK 1.1 version of sunw.io.serializable which inherits from java.io.Serializable. When the serialization code comes to examine the bean it will find that (by transitive closure) the bean inherits from java.io.Serializable. This will allow the bean to be serialized on JDK 1.1

A.3 Rules for a transitional bean developer

In order for a transitional bean to also be usable correctly in a full JDK 1.1 application builder, a bean developer must follow these rules:

- The bean should be delivered in a bean JAR file.
- The bean should use the getFoo/setFoo patterns (see Section 8.3) for properties.
- All of the bean's persistent state must be accessible via getFoo/setFoo properties. This allows application builder tools to save and restore the beans state using the property accessor methods. The one exception to this rule is that the bean can ignore state in java.awt base classes.
- Any events the bean fires should use the new AWT event model and should use the sunw.util.EventObject and sunw.util.EventListener interfaces to mark events and event listeners. However in interacting with AWT, the bean must of necessity use the old AWT event model to catch AWT events.
- If the bean is prepared in the future to use automatic serialization then it must inherit from the sunw.io.Serializable marker interface.
- The bean can't rely on any new 1.1 interfaces.

A.4 Application builder responsibilities

To be able to use transitional beans to create JDK 1.0.2 conformant applets and applications, an application builder tool must:

- Read in a bean from a JAR file.
- It must use its own reflection code to identify properties, events, and public methods, following the rules defined in Section 8.
- Rather than using automatic serialization to store away a bean's state it must use the getXXX methods to read its state and generate some code to call setXXX methods to revive the bean. If the bean inherits from a JDK 1.0.2 java.awt.* component class, then it must generate code to restore the appropriate AWT state.

A.5 Example of a transitional bean

The following three classes define a transitional bean that (a) supports a pair of property accessor methods for getting and setting a simple boolean property called "debug" and (b) fires a beans style event.

Because it inherits from java.awt.Button, it will also automatically get the "label", foreground", "background" and "font" properties that can be inferred from the various methods it inherits.

```
// This class describes the event that gets generated when
// OurButton gets pushed.

public class ButtonPushEvent extends sunw.util.EventObject {
    public ButtonPushEvent(java.awt.Component source) {
        super(source);
    }
}

// This interface describes the method that gets called when
// an OurButton gets pushed.

public interface ButtonPushListener extends sunw.util.EventListener {
    public void push(ButtonPushEvent e);
}

// The OurButton class is a very minimal bean that simply extends the
// standard JDK 1.0.2 Button class to throw a bean event.

public class OurButton extends java.awt.Button implements
                                        sunw.io.Serializable {
    private boolean dbg;
    private Vector listeners = new Vector();

    public OurButton() {
        this("press");
    }

    public OurButton(String label) {
        super(label);
    }

    // Register an Event Listener.
    public synchronized void
    addButtonPushListener(ButtonPushListener bl) {
        listeners.addElement(bl);
    }

    // Remove an Event Listener.
    public synchronized void
    removeButtonPushListener(ButtonPushListener bl) {
        listeners.removeElement(bl);
    }
```

```
    // Catch an old style AWT event and fire it as a new style event.

    public boolean handleEvent(Event evt) {
        if (evt.id == Event.ACTION_EVENT) {

             // Notify each our our listeners.
            Vector   l;
            ButtonPushEvent e = new ButtonPushEvent(this);
            synchronized(this) {
                l = (Vector) listeners.clone();
            }

            for (int i = 0; i < l.size(); i++) {
                ButtonPushListener bl =
                            (ButtonPushListener)l.elementAt(i);
                bl.push(e);
             }

        }
        return super.handleEvent(evt);
    }

    public boolean isDebug() { return dbg; }
    public void setDebug(boolean x) { dbg = x; }
}
```

Appendix B: Future directions

Java Beans 1.0 specifies a complete set of core functionality required for a component architecture including support for properties, events, methods, and persistence. However JavaSoft is also planning to develop some additional APIs to supplement this core functionality.

This section provides an outline of areas where we expect to provide additional APIs in the next release of Java Beans.

B.1 Menubar merging

It should be possible for beans to specify their own menu structure, such that upon activation of the bean that menu structure can be reflected in the user interface of the containing application.

JavaSoft is currently designing such an API. However this will not be available as part of the first release of Java Beans.

B.2 Extended persistence support

In addition to the Java serialization API we plan to add additional APIs to allow objects more detailed control over their persistent state and to provide support for seekable structured storage. The Java serialization APIs were designed with these extensions in mind and include hooks for inter-operating with additional persistence mechanisms.

Appendix C: Change History

Since the release of the initial JavaBeans 1.0 text in October we've made some minor tweaks to add in some small pieces of missing functionality (notably icons), to tighten up some definitions (particularly around manifest files and bound properties), and to fix one potential source of errors (the use of class name strings). There are change bars on the text, but not on the API definitions as these are mechanically generated.

- Added BeanInfo.getIcon method and 4 static constants for different icons. This allows BeanInfo classes to specify icons to represent their bean in toolbars and toolboxes.
- Added support in EventSetDescriptor for MethodDescriptor objects (in addition to Method objects) for listener methods. There is a new EventSetDescriptor constructor taking MethodDescriptor objects as an alternative to Method objects, and you can use getListenerMethodsDescriptors to retrieve MethodDescriptors instead of Methods. This allows BeanInfo classes to provide more detailed information about event listener methods.
- In the various Descriptor methods replaced the use of class name strings with Class objects. This reflects the fact that in Java class name strings are actually context sensitive, so that a class name that makes sense in one piece of code (with one class loader) may be inaccessible to another piece of code that uses a different class loader. Therefore it is better to use Class objects as these have global scope. A new feature in JDK 1.1 "class literals" makes it easier to reference Class objects for specific classes.
- Clarified that property accessor methods may throw checked exceptions (Section 7.3)
- Specified that the JAR manifest file must be called META-INF/MANIFEST.MF (Section 11.3.3).
- Several minor changes to tighten up the specification of bound and constrained properties:
    - Updated Section 7.4.1 to specify that propertyChange events should be fired **after** the property change has been applied
    - Updated Section 7.4.2 to specify that vetoableChange events should be fired **before** the property change has been applied.
    - Updated Section 7.4.3 to specify that for properties that are both bound and constrained you should fire a VetoableChangeListener.vetoableChange event before updating the property and a PropertyChangeListener.propertyChange event after updating the property.
    - Added Section 7.4.4 advising that updates to bound and constrained properties should not fire events if they are simply setting a value that is the same as the current value.

Index of Classes and Interfaces