The Java Series

The JAVA Beans Architecture

The JAVA Component Model

JavaBeans is Java s component model.

Components are self-contained, reusable software units that can be visually combined into composite components and applications using visual application builder tools. JavaBean components are known as *Beans*

In a way, a components model is an extension of OO.

There are other component models:

- ActiveX (OLE + DCOM) is Microsoft s component model

JavaBeans components are implemented in JAVA (portable, etc..).

ActiveX components only run in Windows OSs but are native.

What are components for?

The goal of components models are to:

- Combine components into applications beyond the scope of the language in which they are programmed.
- Build applications using seemingly other people s components manipulating source code as little as possible.

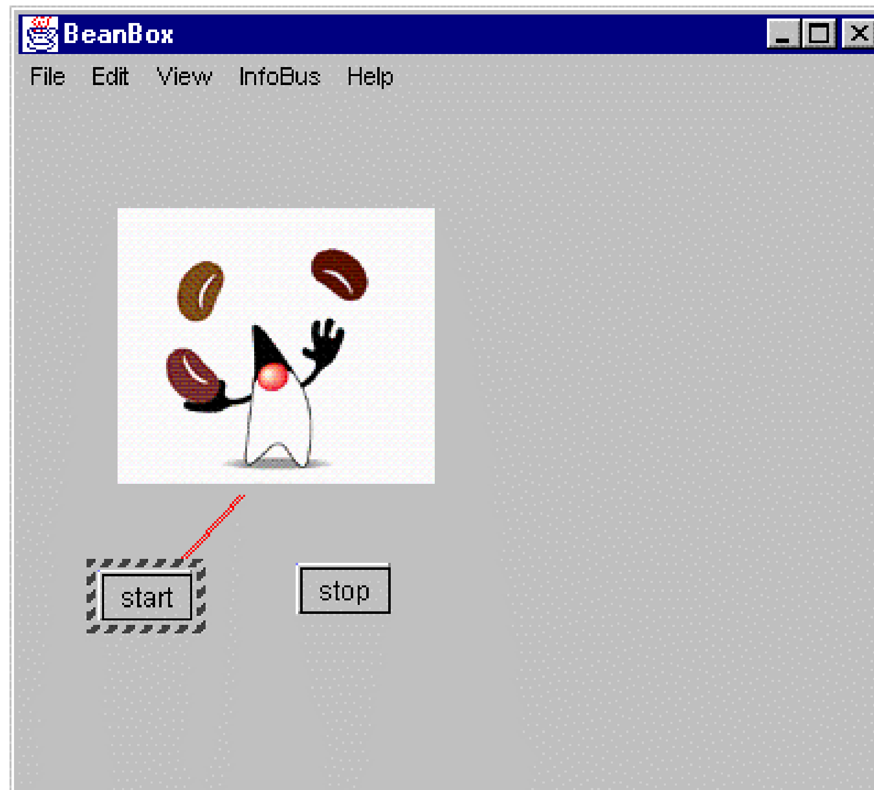
Components expose their features (methods, events, ..) to builder tools for visual manipulation.

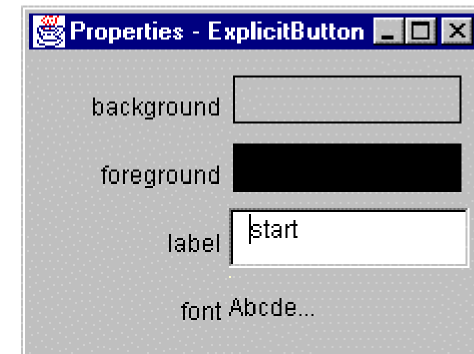
A JavaBeans enabled builder tool (such as IDEs) can examine Beans features and expose them for visual manipulation.

This way we take components written by other people and combine them together in an easy way to write our application.

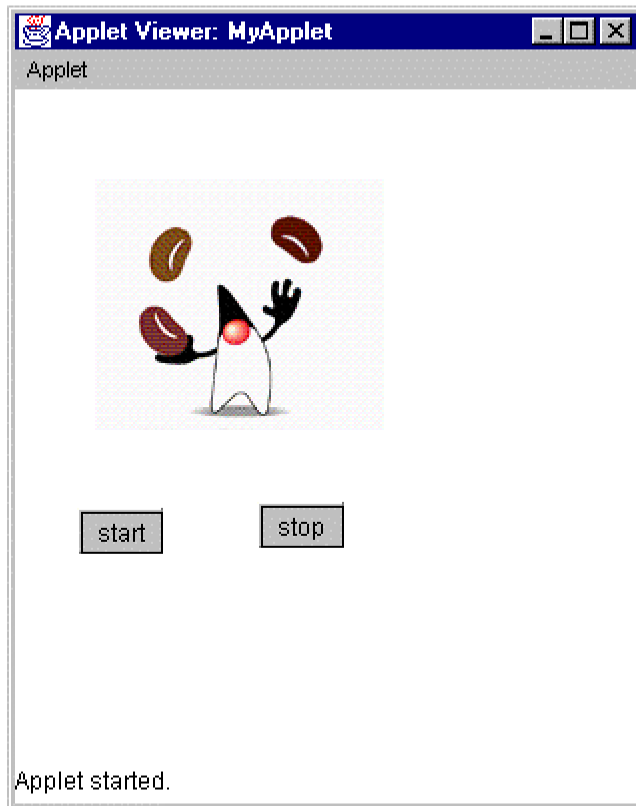
Source code manipulation is replaced by visual design.

Using Components





Using Components



And the building environment creates the application or applet from our design.

With component models:

We can build our own components

Integrate them into any enabled IDE

We are not restricted to factory included components

With component models we create and combine arbitrary components.

Today s aim is to SHOW how this happens

How does it work?

Any component must expose its features.

Any builder must understand how to look for configurable features.

A Component Model is just an specification, a convention for components and builders to understand each other.

A Bean is just a Java object with conforms with the Java Beans specification.

See: **`http://java.sun.com/beans/spec.html`**

Builders following the JavaBeans spec are called BeanBoxes.

What features to expose

A certain component will expose:

- **Properties** that can be manipulated by other components.
- **Events** that the component will launch.
- **Methods** that can be invoked in the component.

For instance, with the builder we can:

- Associate an event in one component with a method in the other component

Reflection

Reflection: Java mechanism to discover dynamically information about any object.

In Java reflection is done through some available classes: Class, Method, Field, etc

For instance, for any object obj:

```
Class c = obj.getClass();
Method[] m = c.getMethods();
for (int i=0; i<m.length; i++) {
  System.out.println("Method: "+m[i].getName());
  Syst …. ("Return value: "+m[I].getReturnType().getName());
```

Design Patterns

A **design pattern** is just a convention on how to name object elements (class names, methods, variables, etc.)

We saw some design patterns in AWT events:

For any EventType we have:

- **<*EventType*>Event** class (ActionEvent, MouseEvent)
- **<*EventType*>Listener** interface (ActionListener, MouseListener)
- **add<*EventType*>Listener** methods (addActionListener, addMouseListener)

Introspection in beans

Introspection: Obtaining information about objects.

Reflection is a form of introspection.

The JAVA Beans specification defines two mechanisms by which components expose their features:

- Using specific design patterns searched through reflection.
- Associating a BeanInfo class which each bean.

When you build a Java component:

- Follow the design patterns of the JavaBeans spec.
- Associate a BeanInfo class with your Bean to describe its features.

When a beanbox encounters a bean:

- Looks for a BeanInfo class or
- Looks for specific design patters.

Event design patterns

To publish and discover events JavaBeans uses the same design pattern as AWT

A certain event is defined in the:

- **<*EventType*>Event** class
- **<*EventType*>Listener** interface

A builder will recognize a component to generate a certain event if it contains these two methods:

- **add<*EventType*>Listener(<*EventType*>Listener)**
- **remove<*EventType*>Listener(<*EventType*>Listener)**

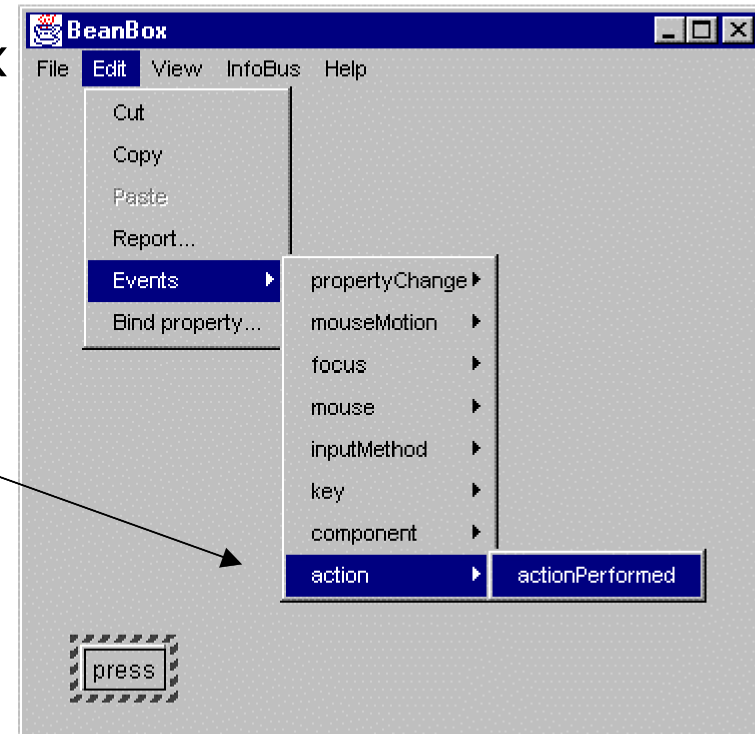
A builder will recognize a component to be able to handle a certain event if it contains any method with that particular event type as argument

Sample

Use Sun s Bean Development Kit. Free from Sun

It contains a simple beanbox which understands the Java Beans specification.

Drop OurButton in a BeanBox

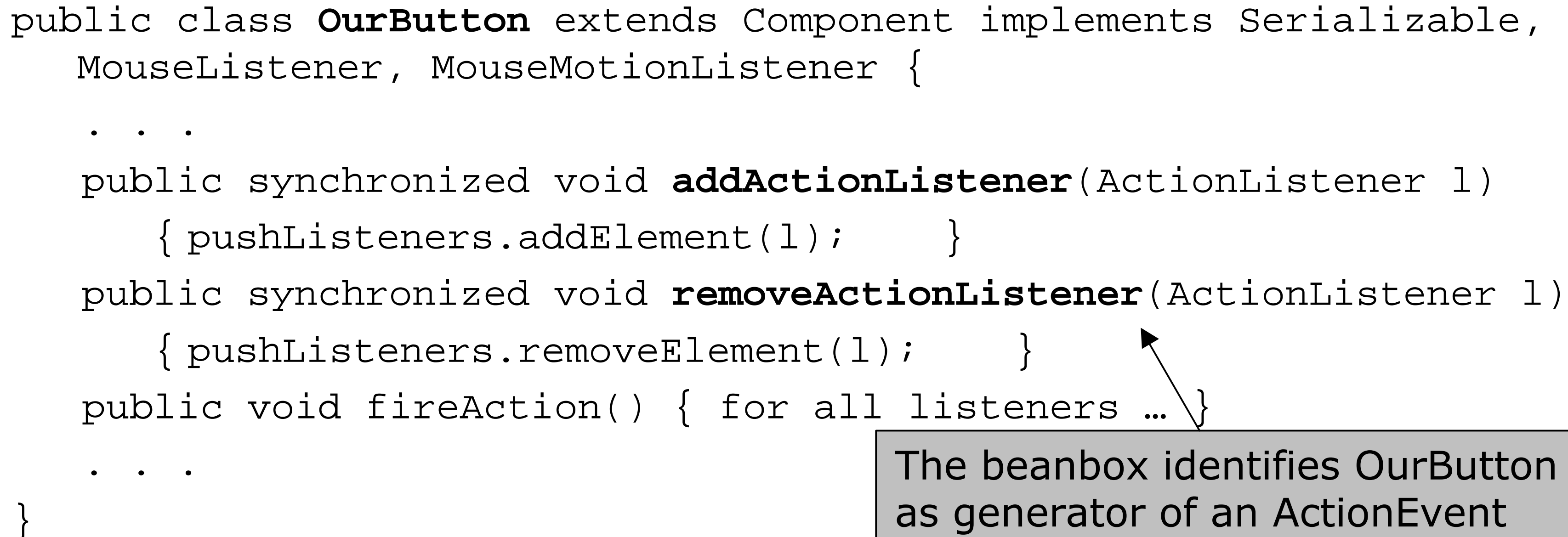


This menu option is generated by the BeanBox by inspecting OurButton component and the correspondent listener class

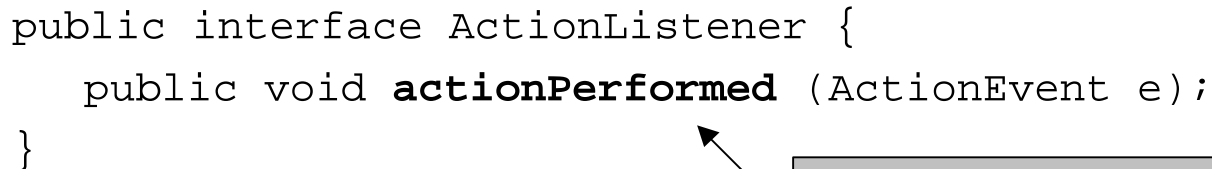
ALL EXAMPLES ARE PART OF THE BDK

Sample

```
Use java.beans.*;
public class OurButton extends Component implements Serializable,
   MouseListener, MouseMotionListener {
    . . .
    public synchronized void addActionListener(ActionListener l)
       { pushListeners.addElement(l);    }
    public synchronized void removeActionListener(ActionListener l)
       { pushListeners.removeElement(l);    }
    public void fireAction() { for all listeners … }
    . . .
}
```

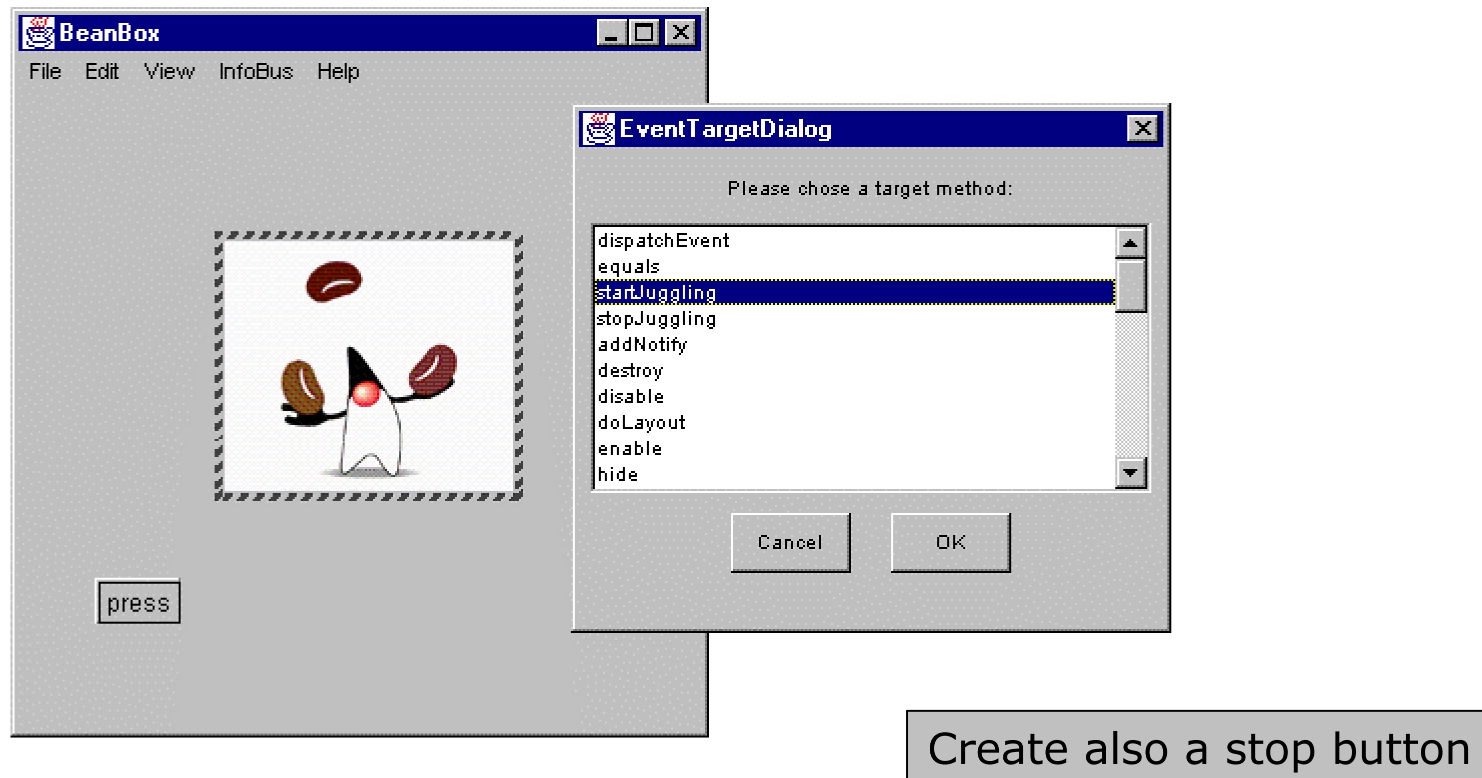



```
public interface ActionListener {
   public void actionPerformed (ActionEvent e);
}
```



Sample

Now, we insert a new bean and link the actionEvent from the button to it



Create also a stop button

Sample

Java creates an adaptor class which implements the actionListener and calls startJugging

An instance of this class is the actual listener.

This is transparent to us. The class is automatically created. This is the way THIS beanbox works.

```
public class ___Hookup_1474c0159e implements
          java.awt.event.ActionListener, java.io.Serializable {

   public void setTarget(sunw.demo.juggler.Juggler t) {
          target = t;
   }

   public void actionPerformed(java.awt.event.ActionEvent arg0)
   {
          target.startJuggling(arg0);
   }

    private sunw.demo.juggler.Juggler target;
}
```

Sample

Note that as creators of the OurButton bean:

We had to implement add/remove methods

This implies maintaining the list of listeners

Our button listens to its own mouse clicks and

invokes the actionPerformed method of all registered listeners.
We must implement in OurButton:

```
public void mouseReleased (MouseEvent e) { … fireAction() … }
public void fireAction() {
  Vector targets = (Vector)pushlisteners.clone();
  ActionEvent actionEvt = new ActionEvent(this, 0, null);
  for (int i = 0; i < targets.size(); i++) {
   ActionListener target =  (ActionListener)targets.elementAt(i);
   target.actionPerformed(actionEvt);
  }
```

But this is generic: works with any actionListener

The Juggler does not need to be an actionListener, since the automatically created adaptor takes care of this function.

Events in BeanInfo

Instead of using introspection through reflection we can also explicitly create an associated class containing that information.

To associate a BeanInfo class with a certain bean just call it **<*BeanClassName*>BeanInfo**

For instance, if we have a bean named **ExplicitButton** the beanbox looks for the **ExplicitButtonBeanInfo** class. If not found then it will use reflection.

If found the beanbox will invoke certain methods to obtain information. For instance, it will expect that

- getEventSetDescriptors returns the list of events generated by the bean.

Events in BeanInfo

If we want to describe the events of our bean through the BeanInfo mechanism:

- We create a class with the appropriate design pattern implementing the BeanInfo interface:

```
public class ExplicitButtonBeanInfo implements BeanInfo
```

- We implement the getEventSetDescriptor method:

```
public EventSetDescriptor[] getEventSetDescriptors() {
  try {
     EventSetDescriptor push = new EventSetDescriptor(beanClass,
       "actionPerformed",java.awt.event.ActionListener.class,
                                "actionPerformed");
     EventSetDescriptor changed = new EventSetDescriptor(beanClass,
       "propertyChange",java.beans.PropertyChangeListener.class,
                                "propertyChange");
     push.setDisplayName("button push");
     changed.setDisplayName("bound property change");

     EventSetDescriptor[] rv = { push, changed};
     return rv;
  } catch (IntrospectionException e) {throw new Error(e.toString());}
```

Reflection vs BeanInfo

With reflection we don t have to worry about anything, just conform with the JavaBeans spec design pattern

With BeanInfo you can control what events to offer to the users of your component. Note the events available in OurButton (inherited through the AWT Component class) and the events available in ExplicitButton, although it inherits from the same class.

Also with the BeanInfo we provide more precise information: labels, event names, class names, icons for the BeanBox, etc

Use the reflection mechanism only when your bean is simple. BeanInfo provides more control.

Using BeanInfo you also must comply with specs.

Manipulating Events

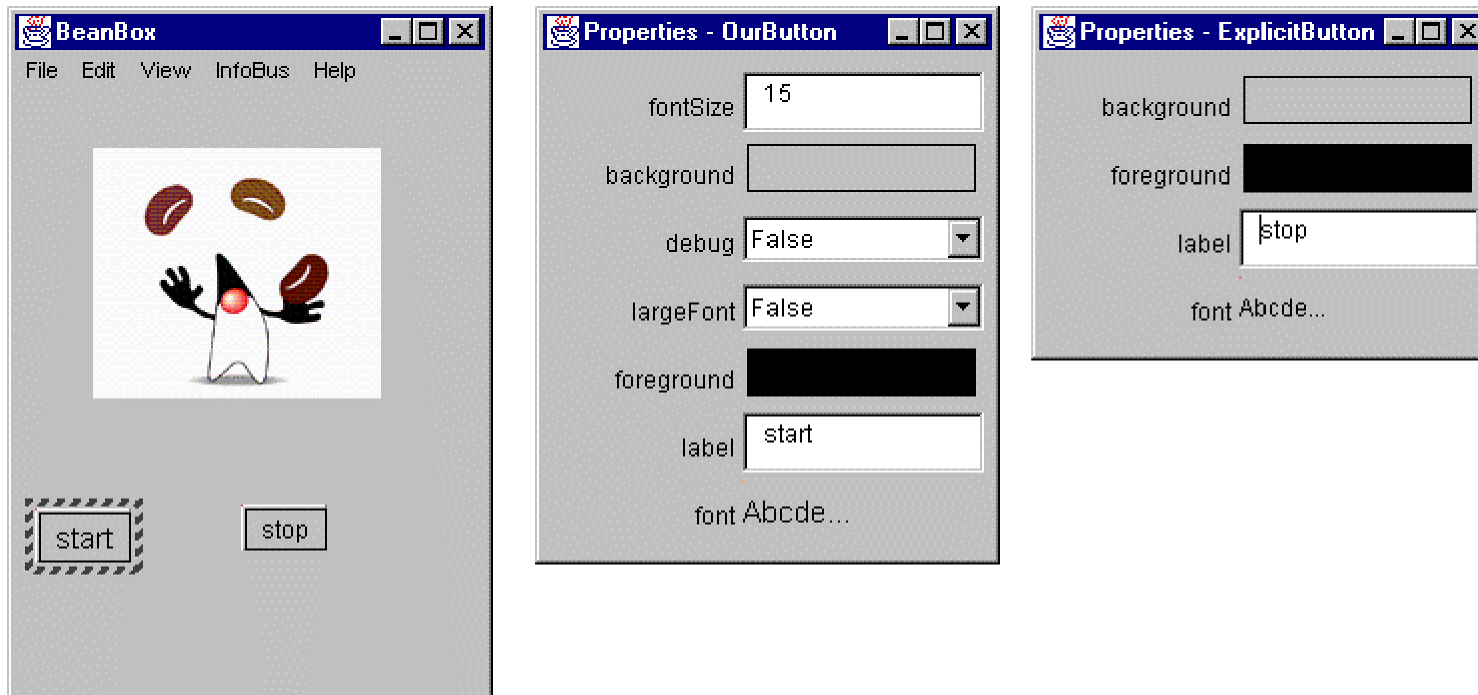
So, to manipulate events we have to:

- Create the appropriate add/remove methods to register listeners on the bean generating the event.
- Whenever the bean generates the event, we create the code to invoke the appropriate method on all listeners.

- Optionally use a BeanInfo class to describe the events.

Bean Properties

Beans expose their properties for manipulation by the user or other components:



As with events, properties can be exposed through reflection or BeanInfo.

Properties through reflection

A property is identified whenever the two following methods exist in a bean class:

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

For instance

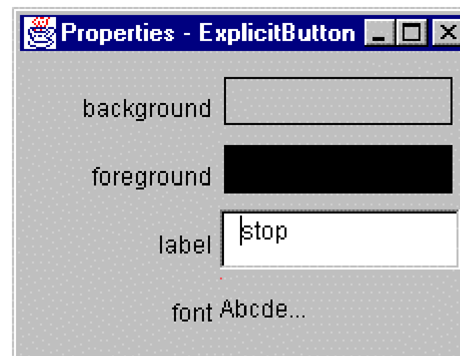
```
private String caption = “press”;
public void setLabel(String newLabel) {
  caption = newLabel;
  repaint();
}

public String getLabel() {    return caption;  }
```

Property editors

For each property type we need a way to edit it so that we can manipulate properties of that type through builder tools.

For each property type (String, Color, Font,   ) the bean box will look for a descendant of the AWT panel named **<*PropertyType*>Editor (StringEditor, ColorEditor, etc.)**

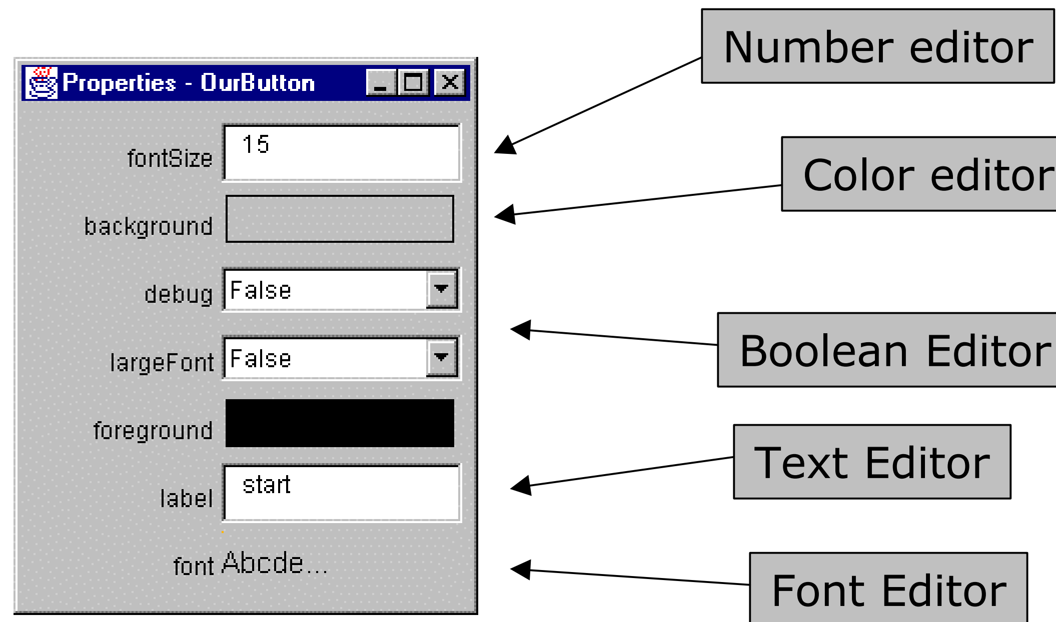
The beanbox builds a property sheet with the properties found and looks for the correspondent property editors:



Property Editors

As with any other calculated class name, the beanbox will look in the classpath for the classes (BeanInfo, Editors, etc.)

The BeanBox provides editors for the most common types of properties.



Property Editors

Any property editor must implement:

a constructor accepting the bean it s controlling.

- **public Object getValue ()** for the beanbox to obtain the value.
- **public setValue (Object)** for the beanbox to set the value, invoking **set<*Property*>** of the bean.
- **public String getJavaInitializationString()** for the beanbox to generate the Java code initializing the property.

This way the beanbox interacts with the property editor.

PropertyChange Events

Any bean can also be a PropertyChangeEvent generator.

This event has the same mechanics as any other event, with add/remove listener methods, a PropertyChangeListener interface, etc.

A bean will trigger a property event whenever it decides that a property change should be notified to the listeners:

```
 public void setLabel(String newLabel) {
   String oldLabel = label;   label = newLabel;
   sizeToFit();
   changes.firePropertyChange("label", oldLabel,   newLabel);
}
```

Manipulating Properties

So the whole process is as follows:

- The beanbox obtains the properties of a certain bean (reflection or BeanInfo).
- The beanbox obtains the property editor for each property (bean spec or BeanInfo).
- The beanbox creates a property sheet putting together the editors for the properties identified.
- The user interacts with the property editors of the property sheet.
- Each property editor will actually change the property of the bean by invoking the setXXX method.
- Each bean may have PropertyChangeListeners to notify about property changes.

Properties and BeanInfo

As with Events, properties can be described in the BeanInfo class associated with a certain bean.

For instance, to specify what are the manipulable properties of a bean, we include the following method in the BeanInfo:

```
public PropertyDescriptor[] getPropertyDescriptors() {
    PropertyDescriptor background = new
        PropertyDescriptor("background", beanClass);
    PropertyDescriptor foreground = new
        PropertyDescriptor("foreground", beanClass);
    PropertyDescriptor font = new
        PropertyDescriptor("font", beanClass);
    PropertyDescriptor label = new
        PropertyDescriptor("label", beanClass);

    PropertyDescriptor rv[] ={background, foreground, font, label};
    return rv;
}
```

Properties and BeanInfo

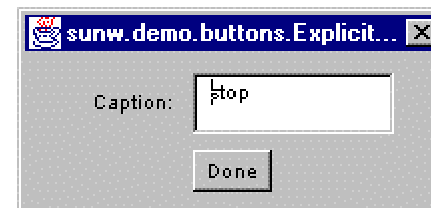
With the BeanInfo class we can also:

- set editors for each property
- set icons
- describe methods to link events to (remember the startJuggling and stopJugling methods)
- set customizers

A Customizer is a Bean Editor

- Sometimes we need more sophisticated ways to edit the properties of a bean (joint editing, etc.)
- A customizer is an AWT panel used to provide a custom property sheet.
- This is the ExplicitButton

customizer



More on Beans

The Beans specification requires Beans to be serializable so that their status can be stored and retrieved.

See save/load options on the BeanBox file menu

All classes related to a certain bean (the bean itself, the bean info, icons, etc.) are bundled within a jar file.

See files generated by the MakeApplet option on the BeanBox file menu.

See where components jar are located.

See a component s jar file.

Enterprise Beans

- Sun is extending the Beans specification into the Enterprise Beans specification.
- Enterprise Beans are regular beans that:
  - Can (de)serialize themselves to/from a database
  - Instances can be accessed in a networked environment through CORBA or RMI. For instance: an enterprise beanbox we will be able to link visually an event in an object to a method invocation on an object on another machine
- The aim is to provide a truly distributed component architecture based on open technologies: beans, CORBA, SQL, etc.

Summary

JavaBeans is just an specification to create easily reusable software components

As components designers, JavaBeans tells us how to publish in an standard way events, properties and methods dealt with by our bean

As IDE designers, JavaBeans tells us how to make components interact with each other so that they can be combined.

As application designers, JavaBeans provide us with a reliable mechanism to put components together, and we worry mainly about integration semantics rather than mangling with source code logistics.

These three aspects are the key issues of the components technology