Overview of JavaBeans Technology

- A JavaBeans component is an object (!)
  - conforms to a communication and configuration protocol (JavaBeans specification).
- The JavaBeans specification prescribes programming conventions and dynamic discovery mechanisms, that
  1. minimize the design and implementation effort for small software components
  2. fully supporting the design, implementation, and assembly of sophisticated components.
- Three fundamental aspects of the JavaBeans components:
  - events, properties, and methods
- "division of labor" strategy.
  - minimal overhead imposed by the framework
  - simplified design and implementation of custom JavaBeans

Component Architecture

- JavaBeans - standard component architecture for Java.
  - JavaBeans API is packaged in java.beans
  - This package includes interfaces and classes that support design and/or runtime operations.
- It's common to separate the implementation into design-only and runtime classes,
  - the design-oriented classes (which assist programmers during component assembly) do not have to be shipped with a finished application.
  - The JavaBeans architecture fully supports this implementation strategy.

Additional Supporting APIs

- The **Glasgow** specs. define the new JavaBeans capabilities.
  - Parts of this specification are incorporated into the Java 2 platform, version 1.2, for example, the drag and drop subsystem;
  - other facilities are available as a Standard Extension, for example, the JavaBeans Activation Framework, which defines standard mechanics for Bean instantiation and activation.
- The **InfoBus** specification defines a secondary API
  - provides another, alternative, communication mechanism among Beans.
  - The InfoBus provides programming conventions and mechanics whereby JavaBeans components can register with either a default or a named "information bus."
  - Components cooperate by getting on the same bus and exchanging information following an asynchronous (event-driven) communication protocol.

Status of Component Technology

- M. D. McIlroy (1968) made a now-historical plea for catalogs of software components.
- The practical tools necessary for McIlroy's vision of libraries of software components now exist,
  - fundamentally with the core Java programming language,
  - JavaBeans API for client-level component assembly,
  - Enterprise JavaBeans specification for server-level component assembly.
- Components have been addressed in a number of languages
  - Smalltalk, Eiffel, and now the Java programming language.
  - The recent shift in programming paradigms, attributable in part to Internet developments, has forced component technology out of the shadows.

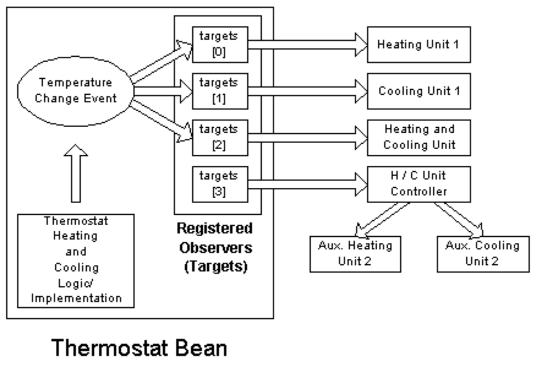
Bean Behavior

- Component technology in the 1990s incorporates the event-driven methodology of the late 1980s.
  - synchronous communication via method calls.
  - components communicate asynchronously using an event and notification model
- **Subject-observer or source-target** communication pattern
  - Beans are source objects.
- A Bean occasionally sends notifications of changing state to all registered targets.
  - notifications are component-specific;
  - signal the occurrence of one or more significant events in the Bean instance.
  - In a drop-down list, for example, selecting an item would constitute such an event.

AP 6/02

Typical Bean Interactions

- Often, JavaBeans component will function as a source for certain types of events, yet be capable of registering as a target for events produced by other components.



AP 6/02

The JavaBeans API

- The JavaBeans API includes several interfaces and classes in the java.beans package.
- Interfaces and classes from other Java technology API areas:
  - The Java event model: java.util.EventObject, java.awt.event
  - Object serialization: java.io.Serializable, java.io.Object*
  - Reflection: java.lang.reflect
- JDK 1.1 introduced subject-observer (source-target) event model.
  - JDK 1.1 provides base-level support for this event model outside the AWT package, specifically, in the java.util package.
  - The relevant interface, class, and exception are java.util.EventListener, java.util.EventObject, and java.util.TooManyListenersException.

AP 6/02

Object Serialization

- Prerequisite for basic JavaBeans functionality.
  - When programmers assemble, configure, and connect Beans using an IDE, the JavaBeans components must be "live," dynamically created objects.
  - The IDE must be able to save the worksheet's state at the end of the day and restore it at the start of a subsequent session.
  - That is, a Bean's state must persist via external storage.
- JavaBeans must implement the Serializable interface.
  - Serializable is a tagging interface; that is, it marks an object as suitable for serialization by the Java runtime environment
  - All class/instance variable are saved (except those marked transient)
- Before attempting to write an object to disk the Java interpreter verifies that the object implements Serializable.

AP 6/02

Reflection

- Reflection is the third indispensable API (java.lang.reflect) for the JavaBeans architecture.
  - With reflection, it's straightforward to examine any object dynamically, to determine (and potentially invoke) its methods.
- IDE examines a Bean dynamically to determine its methods,
  - analyze design patterns in method names and put together a list of access methods that retrieve and set instance data,
  - for example, **getForeground()** and **setForeground()** for retrieving and setting **foreground** color.
  - An instance/state variable with this type of access methods is called a property.
- IDE uses reflection to determine the Bean's properties
  - presents them for editing in a graphical window, (property sheet).
  - By using standard naming conventions a programmer can design a Bean that's configurable in a graphical builder tool.

JavaBeans Design Issues

- JavaBeans objects are like other user-defined data types, but with the following additional options that make the objects more useful:

- Providing a public no-argument constructor
- Implementing java.io.Serializable
- Following JavaBeans design patterns
  - Set/get methods for properties
  - Add/remove methods for events
  - Java event model (as introduced by JDK 1.1)

- Being thread safe/security conscious
  - Can run in an applet, application, servlet, ...

Design Issues (contd.)

- For an IDE to instantiate a bean, the class implementation must provide a no-argument constructor.
- For an IDE to automatically present various state variables for configuration/editing,
  - there must be access methods that follow prescribed naming, return value, and signature conventions—the JavaBeans design patterns.
- This design pattern principle applies to events as well.
  - For an IDE to allow communication connections between Beans, there must be add and remove methods that the IDE can invoke to register and unregister targets (Beans that listen to and respond to event notifications).
  - An IDE must be able to connect the event notifications from one Bean to the event-handling functionality of another Bean.

JavaBeans Event Model

- JavaBeans uses the Java event model to communicate.
  - Events provide an alternative to (synchronous) method invocations for any type of communication between components in which "background notifications" are appropriate.
  - Components providing one or more computational services can acknowledge and handle other services on an event-driven, or asynchronous, or "logical interrupt" basis.
- In an event-driven paradigm, the source and target orientation is a matter of context.
  - A component can be a source for one type of event and a target for another.
  - With JavaBeans, you're almost always implementing some type of source functionality—for significant events such as temperature changes, progress-bar state changes, and so on.

Event Handling

Event-driven designs are ideal for a variety of tasks:

- Handing user interface events:
  - Mouse actions
  - Keyboard events
- Managing/reporting inter-client connections:
  - JDBC Bean that connects to database server
  - Notifies a client of specific changes in a database
  - Accepts database requests and notifies a client when the data is available
- Other events:
  - Property changes in a Bean
  - Any general-purpose notification from one object to another

- The event notification process passes event-related data to each registered target in a specialized event object.
  - java.util.EventObject is the base class for event objects.

Event Listeners

- For a Bean to notify a target via the prescribed method call(s), the Bean must have a reference to the target.
- Beans support target registration and unregistration with add/remove methods:

```
– public void addAnEventListener(AnEventListener x);
– public void removeAnEventListener(
                              AnEventListener x);
```

AWT brings many predefined Event Types

```
public interface abstract java.awt.event.ActionListener
    extends java.lang.Object, implements java.util.EventListener {
    public abstract void actionPerformed (  java.awt.event.ActionEvent )
}

public synchronized class java.awt.event.ActionEvent
    extends java.awt.AWTEvent {
    public static final int ACTION_PERFORMED = 1001;
    public static final int SHIFT_MASK = 1;
...
    public java.awt.event.ActionEvent (java.lang.Object, int, java.lang.String);
    public java.awt.event.ActionEvent (java.lang.Object, int, java.lang.String, int);
    public java.lang.String getActionCommand ();
    public int getModifiers ();
    public java.lang.String paramString ();
}
```

Example: Event Source

```
import java.io.Serializable;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;

public class EventClockBean implements Serializable {
    protected ActionListener listener;
    public void addActionListener( ActionListener ae ) { listener = ae;}
    public void removeActionListener( ActionListener ae ) { listener=null;}
...
    if (listener != null) {
        ActionEvent ae = new ActionEvent(this,0,time);
        listener.actionPerformed( ae );
    }
```

Example: Event Sink

```
import java.awt.Label;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.io.Serializable;

public class ActionLabel extends Label
    implements ActionListener, Serializable {

    public ActionLabel() {}
    public void actionPerformed( ActionEvent ae ) {
        setText( ae.getActionCommand() );
    }
}
```

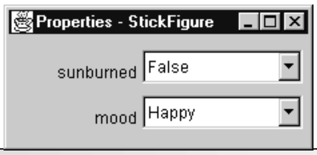
Multiple Event Sinks

```
private Vector targets = new Vector();
public synchronized void addStickFigureListener(StickFigureListener l) {
    targets.addElement(l);
}
public synchronized void removeStickFigureListener(StickFigureListener l) {
    targets.removeElement(l);
}
...
protected void notifyTargets() {
    Vector l;
    StickFigureEvent s = new StickFigureEvent(this);
    synchronized(this) { l = (Vector) targets.clone(); }
    for (int i = 0; i < l.size(); i++) {
        StickFigureListener sl = (StickFigureListener) l.elementAt(i);
        sl.stickFigureChanged(s);
  }
}
```

Access to Vector is synchronized

Properties

- Properties are the public attributes of a Bean that affect its appearance or behavior,
  - for example, background color, foreground color, font, and so on.
  - For a thermostat Bean, the temperature change notification interval might be designed as an integer property, say, one degree Celsius or three degrees Fahrenheit.
  - For a stick-figure Bean, whether or not the Bean instance is sunburned could be a boolean property.
- IDEs typically present properties in a property sheet (dialog box) for editing:



sunburned False

mood Happy

AP 6/02

Simple Properties

- Properties are determined from get/set access method combinations that follow a prescribed naming convention:

```
public void setXXX(TYPE value);
public TYPE getXXX();
```

- The name of the property is the common part of the get/set method names, that is, the characters following "get" or "set".
  - For the StickFigure Bean, **mood** (happy or sad) could be a property:

```
public void setMood(int mood) {
    this.mood = mood; repaint();
}
public int getMood() {
    return mood;
}
```

AP 6/02

Indexed Properties

- Besides simple properties, the JavaBeans model supports indexed properties.
  - Naming conventions for access methods:

```
public void setXXX(int index, type value);
public type getXXX(int index);
public void setXXX(type values[]);
public type[] getXXX();
```

- One example of a property that fits this model is color values:

```
public void setColorTable(int index, Color value);
public Color getColorTable(int index);
public void setColorTable(Color values[]);
public Color[] getColorTable();
```

Bound and Constrained Properties

- Variations on standard properties: .
  - Bound properties support the registration and notification of "interested parties" whenever the value of the property changes.
  - Constrained properties take this notification model one step further, allowing the notified party to exercise a veto, to prevent the property change.
- Unlike with event handling, most of the functionality required to support bound and constrained properties is handled by the JavaBeans framework.
- Bound properties are useful for Beans that want to allow instances of the same Bean class or some other Bean class to monitor a property value and change their values accordingly (to match the "trend setting" Bean).
  - For example, consider a GUI Bean that wants to allow other GUI Beans to monitor a change in its background color to update their backgrounds accordingly.

Implementing a Bound Property

- Bean class must instantiate an object in the JavaBeans framework that provides the bulk of bound property‘s functionality,
  - Bean must implement registration and unregistration methods that simply invoke the appropriate methods in the JavaBeans framework.

```
private PropertyChangeSupport changes =
    new PropertyChangeSupport(this);

public void addPropertyChangeListener(
    PropertyChangeListener p) {
          changes.addPropertyChangeListener(p);
}
public void removePropertyChangeListener(
    PropertyChangeListener p) {
          changes.removePropertyChangeListener(p);
}
```

Bound Properties (contd.)

- Then, each bound property must invoke the firePropertyChange() method from its set method:

```
    public void setMood(int mood) {
          int old = this.mood; this.mood = mood;
          repaint();
          changes.firePropertyChange("mood",
             new Integer(old), new Integer(mood));
    }
```

- At this point, the PropertyChangeSupport object takes over and handles the notification of all registered targets.
  - Note that PropertyChangeSupport provides general-purpose functionality following a prescribed protocol.
  - Specifically, the method invocation for firePropertyChange() must provide the property name, as well as old and new values, which are passed along to notified targets.

Bound Properties (contd.)

- The listener (target object) must provide a propertyChange() method to receive the property-related notifications:

```
public void propertyChange(PropertyChangeEvent e) {
  // ...
}
```

Constrained Properties

- add the functionality that the notified listener can object to the property change and execute a veto.
  - To support constrained properties the Bean class must instantiate the a **VetoableChangeSupport** object, and implement the corresponding registration-related methods:

```
private VetoableChangeSupport vetoes =
     new VetoableChangeSupport(this);
public void addVetoableChangeListener(
     VetoableChangeListener v) {
     vetoes.addVetoableChangeListener(v);
}
public void removeVetoableChangeListener(
     VetoableChangeListener v) {
     vetoes.removeVetoableChangeListener(v);
}
```

VetoableChange

- The set method for bound-constrained properties is slightly more complicated:

```
public void setMood(int mood)
    throws PropertyVetoException {
       vetoes.fireVetoableChange("mood",
              new Integer(this.mood), new Integer(mood));
       int old = this.mood;
       this.mood = mood;
       repaint();
       changes.firePropertyChange("mood",
              new Integer(old), new Integer(mood));
}
```

VetoableChange (contd.)

- Specifically, the set method must accommodate the exception PropertyVetoException. Also, the sequence of operations is:
  1. Fire the vetoable change notification
  2. Update the appropriate state variables
  3. Fire the standard property change notification, if bound
- A veto-interested target object must implement the vetoableChange() method:

```
public void vetoableChange(PropertyChangeEvent e)
       throws PropertyVetoException {
       // ...
}
```

- It exercises a veto by (1) including a throws clause for PropertyVetoException and (2) raising the exception (throw new PropertyVetoException();), as appropriate.

Introspection and BeanInfo

- The Java programming language is dynamic.
  - A class instance "knows" its data type, the interfaces it implements, and the data types of its instance variables.
  - An object can discovery many things about objects for which it has a reference, for example, an object's methods and the methods' parameters and return types.
  - With this information, an object can instantiate an object and formulate a method call on the fly (higher flexibility than source code-level access)
- Introspection: the process of discovering an object's characteristics
  - The JDK provides a collection of classes and interfaces for introspection and dynamic manipulation of objects, commonly known as the Reflection API.
  - Reflection is one of the core Java APIs and is packaged in java.lang.reflect.

Reflection API

- Very general, low-level examination of objects.
  - The JavaBeans framework provides a higher level class, Introspector, that's used by an IDE when working with Beans.
  - An Introspector object assists in discovering a Bean's configurable characteristics.
  - Developers who use the JavaBeans architecture don't typically directly use Introspector, but their IDE environment does use it.
- The Introspector class provides functionality for a container to discover information about a Bean,
  - either by directly querying the Bean or from
  - working with a complementary Bean configuration class that optionally accompanies each Bean.

BeanInfo

- Complementary, support class is called a bean-info.
  - The JavaBeans framework provides the interface **BeanInfo**,
  - describes the services that bean-info classes implement,
  - for example publishing the list of configurable properties or defining an alternative way of specifying accessor methods.
  - An Introspector object manipulates and makes available a Bean's configuration services in a general-purpose manner using the BeanInfo interface.
  - When there is no bean-info class, the Introspector object uses reflection to discover a Bean's properties.

BeanInfo (Contd.)

- There are a variety of configuration possibilities with Beans:
  - properties, property editors, custom configuration dialog boxes, and so on.
  - A Bean publishes its configuration support via methods in its bean-info class.
  - A Bean analyzer then instantiates the bean-info class and queries the appropriate method during the Bean configuration process.
- A Bean analyzer searches for a bean-info class by appending "BeanInfo" to the Bean's class name, for example,
  - MyWidgetBeanInfo
  - TextFieldBeanInfo
  - StickFigureBeanInfo
- Each IDE is free to design its own Bean analyzer class(es), but in all cases the operation would be similar to:

```
TextField tf = new TextField();
  BeanInfo bi = Introspector.getBeanInfo(tf.getClass());
```

Working with BeanInfo

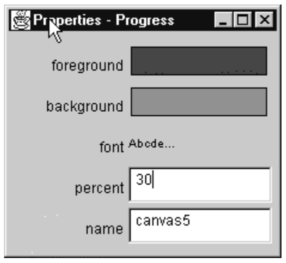
- At times, no bean-info class is required;
  - it's sufficient to provide standard, bound, and constrained properties following the naming conventions outlined previously.
- At other times, it's sufficient to provide one or two configuration specifications,
  - for example, to restrict the number of properties displayed in the property sheet or provide a custom property editor.
  - For a StickFigure Bean, it might be important to provide a drop-down list for setting the mood property.
- As a convenience for the developers who use the JavaBeans architecture, the JavaBeans API provides **SimpleBeanInfo**,
  - a class that implements BeanInfo with empty-body methods.
  - You simply override the appropriate methods with implementations that build and return the appropriate configuration data.

StickFigureBeanInfo

```
import java.beans.*;
public class StickFigureBeanInfo extends SimpleBeanInfo {
  public PropertyDescriptor[]
      getPropertyDescriptors() {
    try {
      PropertyDescriptor pd1 = new
        PropertyDescriptor("mood", StickFigure.class);
      pd1.setBound(true);
      pd1.setPropertyEditorClass(MoodEditor.class);
      PropertyDescriptor pd2 = new
        PropertyDescriptor("sunburned", StickFigure.class);
      pd2.setBound(true);
      return new PropertyDescriptor[] {pd1, pd2};
    } catch (Exception e) {
      return null;
}}}
```

Custom Property Editors

- IDEs provide property sheets for editing a Bean's configurable state.
  - Property sheets vary from one IDE to another, but typically appear as a top-level dialog box.
- Consider the BeanBox's property sheet for a Progress Bean with no bean-info class:



Custom Property Editors (contd.)

- The JavaBeans framework provides a general-purpose mechanism for supplying custom property editors of specific designs.
  1. Extend the PropertyEditorSupport class
  2. Implement the getValue() and setValue() methods
  3. Implement the getAsText() and setAsText() methods
  4. Implement the getTags() method for displaying values in a drop-down list
- The getValue() and setValue() methods are invoked by the framework and provide a way to display and update values.
  - getAsText() and setAsText() methods map discrete values to user-friendly strings.
  - getTags() lists the strings (tags) for the drop-down list.

Custom Property Editor for StickFigure

```
public class MoodEditor extends PropertyEditorSupport {
   protected int mood;
   public void setValue(Object o) {
     mood = ((Integer)o).intValue(); firePropertyChange();
   }
   public Object getValue() { return new Integer(mood); }
   public String getAsText() {
     switch (mood) {
       case StickFigure.HAPPY:
                    return StickFigure.HAPPY_STR;
       case StickFigure.SAD: return StickFigure.SAD_STR;
       case StickFigure.AMBIVALENT:
                    return StickFigure.AMBIVALENT_STR;
       default: return StickFigure.HAPPY_STR;
   }}
```

SetAsText() and getTags()

```
public void setAsText(String s) throws IllegalArgumentException {
    if (s.equalsIgnoreCase(StickFigure.HAPPY_STR))
      mood = StickFigure.HAPPY;
    else if (s.equalsIgnoreCase(StickFigure.SAD_STR))
      mood = StickFigure.SAD;
    else if (s.equalsIgnoreCase(StickFigure.AMBIVALENT_STR))
      mood = StickFigure.AMBIVALENT;
    else
      mood = StickFigure.HAPPY;
    firePropertyChange();
  }
  public String[] getTags() {
    return new String[] {
      StickFigure.HAPPY_STR, StickFigure.SAD_STR,
      StickFigure.AMBIVALENT_STR};
  }
```

Custom Property Editors (contd.)

- Declarative specification for a custom property editor,
  - developer off-loads much of the work onto the JavaBeans framework.
  - In many cases, this approach is considerably easier for the programmer than directly building the actual user interface for a property editor.
- Publish the property editor via the bean-info class:

```
public PropertyDescriptor[] getPropertyDescriptors() {
PropertyDescriptor pd1 =
    new PropertyDescriptor("mood", StickFigure.class);
pd1.setBound(true);
pd1.setPropertyEditorClass(MoodEditor.class);
...
return new PropertyDescriptor[] {pd1, ...};
```

Customization Dialogs

- Sometimes the developer who uses the JavaBeans architecture simply needs total freedom to design a property editor for one or more, possibly specialized, properties.
  - In this case, the JavaBeans framework allows the developer to design and register a custom, graphical object—typically, as a collection of GUI components in a container (panel).
  - Most IDE's display the panel inside a dialog box. In some IDEs, the dialog is modal; in others, it resides on the desktop as a top-level window.
- This customizer implementation is free to use any of the classes provided by the JavaBeans framework (e.g.; PropertyEditor).
  - The customizer must specialize java.awt.Component and implement java.beans.Customizer.
  - A Bean analyzer uses the BeanInfo-prescribed method getCustomizerClass() to retrieve the customizer.