Introduction to JavaBeans

Creating Reusable Software Components in Java

Presented by Eric Burke for the
St. Louis Java Special Interest Group
July 10, 1997

What is a Software Component?

- A reusable piece of software that:
  - has a well-specified external interface
  - can be used in unpredictable combinations
  - is a standalone entity
    - not a complete application
    - typically more than a simple class or function
      - a JavaBeans component can be created by writing a single class, however this class must conform to various coding guidelines

Example Software Components

- GUI Components:
  - Button, Label, Grid, Window, Menu
- Non-visual Components:
  - Database connector component
  - Stopwatch/Clock for simulations

Problems with Non-Java Components

- Non-portable (ActiveX, Motif)
- Often difficult to integrate with GUI builder tools
  - ActiveX components are easy to use in Windows tools only
  - adding Motif widgets to UNIX tools usually requires re-linking the builder tool itself
- Current component models are not fully object-oriented
  - you cannot simply subclass a component to customize its behavior

What is a Java Bean?

- “A Java Bean is a reusable software component that can be visually manipulated in builder tools.” - JavaBeans FAQ
- Key Technologies:
  - Introspection
  - Customization
  - Events
  - Properties
  - Persistence

JavaBeans Terminology

- JavaBeans
  - an API specification for creating reusable software components in Java
- Bean
  - a Java class that conforms to the JavaBeans conventions
- Properties
  - attributes of a Bean, such as textColor or width
- Property Sheet
  - a grid of property name and value pairs that allows customization of components in a builder tool
- Events
  - provide notification to other classes that something about the Bean has changed

JavaBeans Terminology Cont'd

- Introspection
  - the process of analyzing a Bean to determine what capabilities it has
- Customization
  - setting properties on a Bean
  - users customize properties in builder tools
- Persistence
  - storing the state of an object to secondary storage, usually to a file or database

JavaBeans Goals

- Compact and Easy
  - you can begin creating Beans with minimal features, adding advanced features as needed
  - no common base class such as java.beans.Everything (other than java.lang.Object)
- Portability
  - all Beans are 100% pure Java
    - this means that JavaBeans components are portable to every platform that supports JDK 1.1 or better
- Use Existing Java Language Features
  - a Bean is self-describing, requiring no external description file

Using a Beans Application Builder

- Import custom Beans from JAR files
- Select Beans from component palette, visually arrange them
- Customize Beans through property sheets
- Drag & Drop connections between Beans
- Test the application

Minimal Bean Characteristics

- Implements java.io.Serializable or java.io.Externalizable
  - do not store references to other Beans
- Follows JavaBeans Design Patterns
  - get/set methods to access properties
  - add/remove methods to register event listeners (JDK 1.1 Event model)
- Has a public no-arg constructor
- Is thread safe
  - all Beans must work in a multithreaded application
- Is subject to the Java Applet security model
  - assume that the Bean is running in an untrusted applet

Properties

- Simple Properties
  - read only, write only, or read/write
- Bound Properties
  - provide notification when the property changes
  - java.beans.PropertyChangeSupport makes it easy to create bound properties
- Constrained Properties
  - listeners may veto proposed changes
  - java.beans.PropertyVetoException
  - java.beans.VetoableChangeListener
- Indexed Properties
  - allow for arrays of values to be set or retrieved

Bound Property Example

```
// PropertyChangeSupport is used to notify
// listeners when properties change
private PropertyChangeSupport changes =
    new PropertyChangeSupport(this);

// label is a bound property
public void setLabel(String newLabel) {
    String oldLabel = this.label;
    this.label = newLabel;
    changes.firePropertyChange(
        “label”, oldLabel, newLabel);
}

// this is how listeners register
public void addPropertyChangeListener(
    PropertyChangeListener l) {
    changes.addPropertyChangeListener(l);
}
```

Design Patterns for Properties

```
// simple or bound property “label”
void setLabel(String label)
String getLabel()

// indexed property “count”
void setCountAt(int n, int count)
int getCountAt(int n)

// constrained property “duration”
void setDuration(int duration)
  throws PropertyVetoException
```

Events

- Beans must conform to the JDK 1.1 style event model
- Event Adapters:
  - provide the generic “wiring” between sources and listeners
  - conform to the particular EventListener interface expected by the event source

Example Event Adapter

```
// adapter to map “OK” button events
// to “okAction()” method
class OKAdapter implements ActionListener {
  private MyDialog dialog;
  public OKAdapter(MyDialog dest) {
    dialog = dest;
  }
  public void actionPerformed(
        ActionEvent e) {
    dialog.okAction();
  }
}

public class MyDialog extends Dialog {
  private Button okButton
    = new Button(“OK”);

  public MyDialog() {
    okButton.addActionListener(
        new OKAdapter(this));
  }

  // remainder of code omitted...
}
```

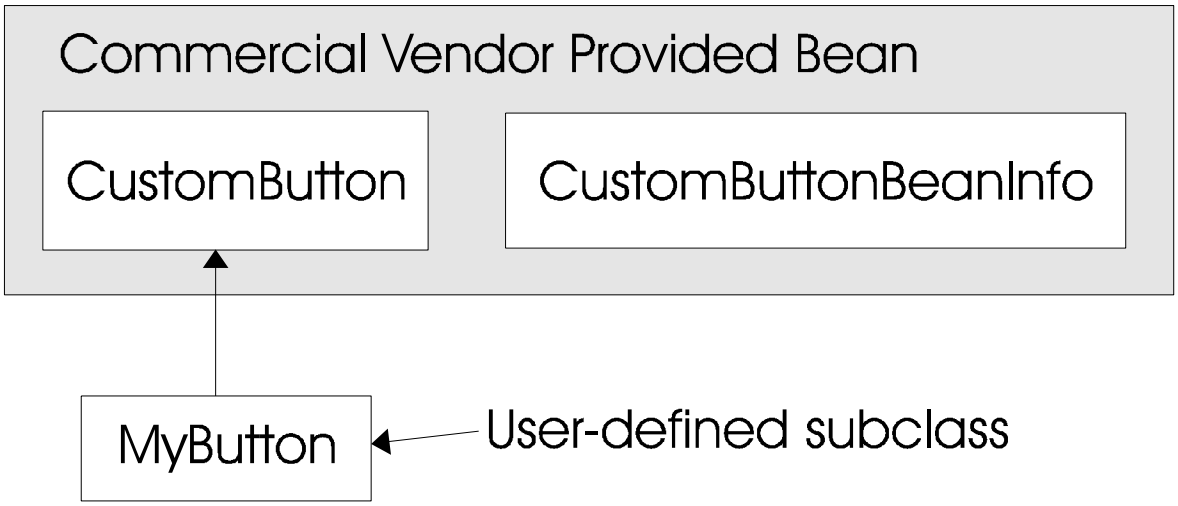
Event Adapter Proliferation

- Real applications may have MANY event adapters
- Anonymous inner classes may be used, but you still end up with a lot of .class files
- Application builder tools may use reflection
  - instead of generating a unique class to respond to an event, builder tools may use a Hashtable to map event sources to target methods
  - this technique is not recommended for hand-coding because you lose strong type-checking at compile time

Introspection

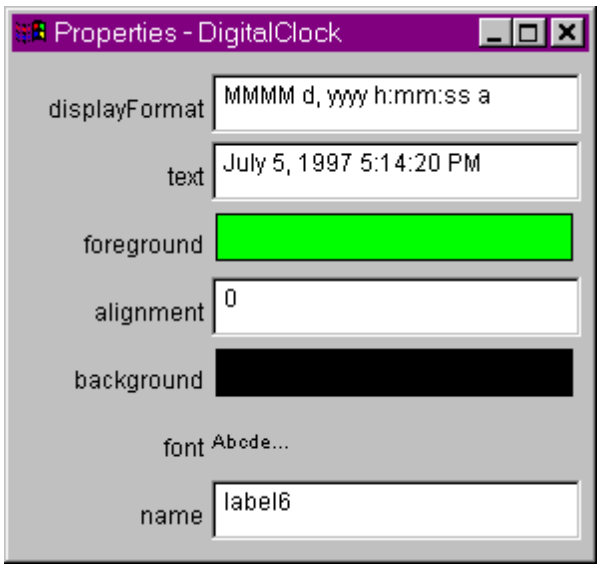
- java.beans.Introspector
  - Locates BeanInfo object
  - Decapitalizes Strings
    - infers property or event names when explicit BeanInfo is not present
  - Typically invoked by application builder tools
- java.beans.BeanInfo
  - Provides access to information about a Bean: Events, Properties, Methods, Icons
  - May be explicitly provided by the Bean author

How Introspection Works



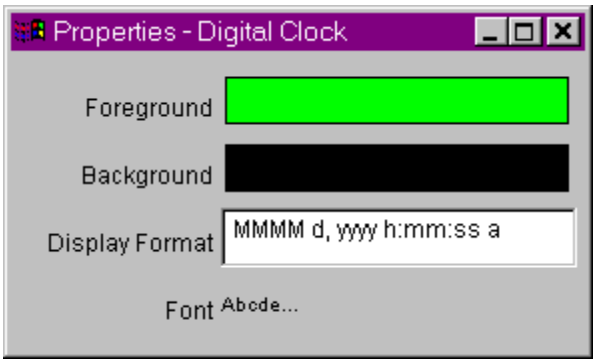
- java.beans.Introspector sees that MyButton has no explicit BeanInfo class
- low-level reflection is used to analyze “design patterns” in MyButton
  - see Appendix B
- parent class is analyzed, and its BeanInfo is located and added to information found in MyButton
- introspection stops, because explicit BeanInfo was found

BeanInfo at Work



Without a BeanInfo class, the Introspector class obtains a lot of information about a Bean

But BeanInfo can make your Bean look much more professional...



- The Bean name can be specified
- Only specific properties are visible
- Property names are capitalized
- Custom editors can be specified

Customization

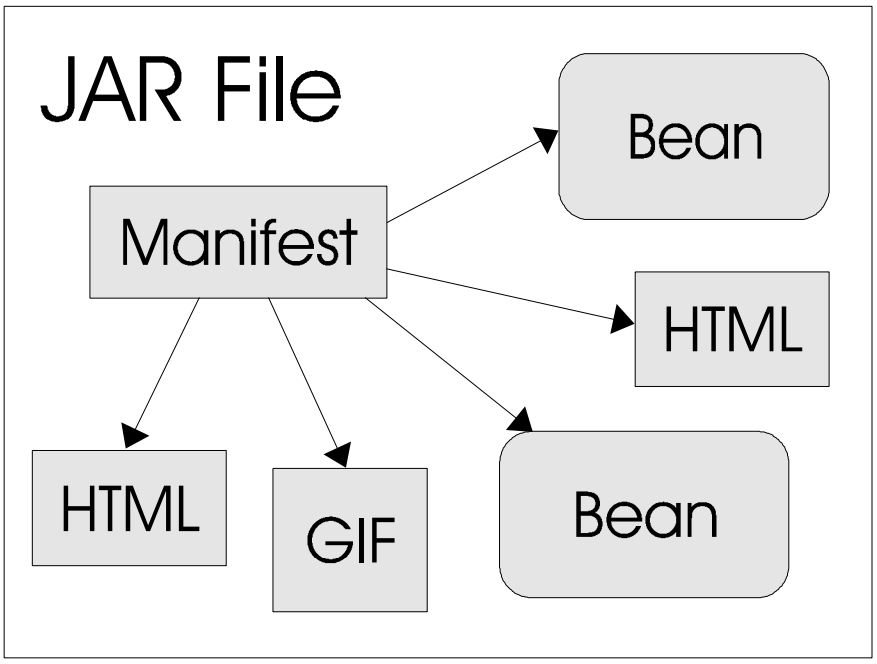
- Property Sheets allow users to customize Bean properties
- java.beans.Customizer
  - allows Bean authors to create customizers as an alternative to property sheets
  - customizers are only used by application builder tools
    - these custom editors can be anything ranging from a trivial customizer to a full-blown “wizard”
- Individual Bean properties may be edited via default editors, or the BeanAuthor can provide a custom property editor

JAR File Contents

- An optional Manifest file describes the contents of the JAR file
- .class files for one or more Beans
- Images, usually stored in GIF format
- HTML Help files
  - HTML files can be embedded in the JAR file to serve as documentation for the Beans
    - HTML 2.0 format
    - top-level documentation should be stored in the JAR file as <locale>/<bean-name>.html
- Resource files
  - this is “open-ended” according the the JavaBeans specifications, such as images, sound, video, property file, or whatever

Packaging

- All Beans are delivered in JAR Files
- A JAR file is a ZIP format archive file containing one or more Beans and various related files



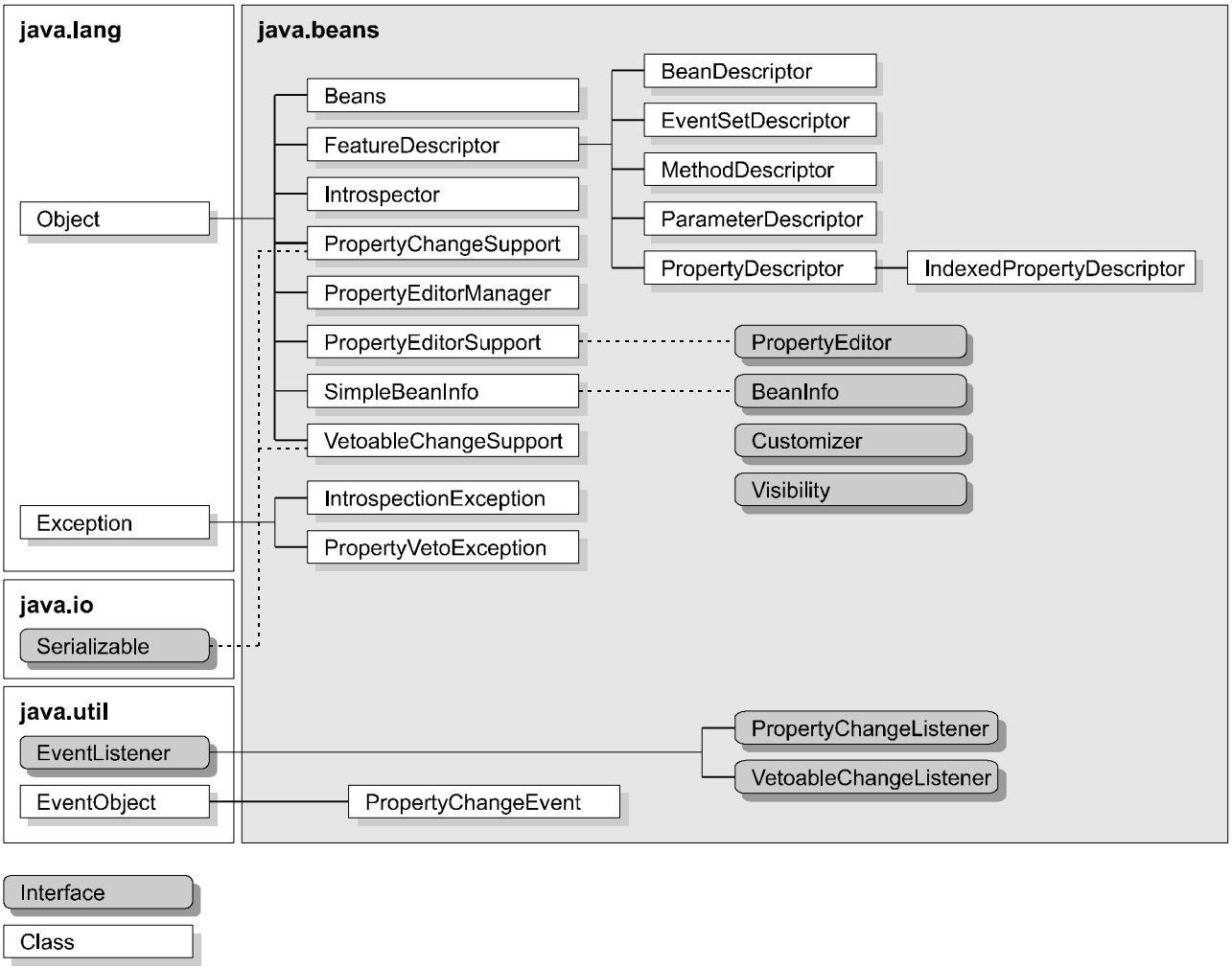
Beans Development Kit

- Provides example source code for several simple Beans
- Includes HTML documentation for the JavaBeans API
- An HTML tutorial is included
- An application called BeanBox is included
  - allows developers to test Beans
  - demonstrates how builder tools will use introspection to learn about Beans
  - BeanBox is NOT a complete builder tool

Summary

- JavaBeans provides a superior model for software component reuse
- Applications will be easier to construct as new Beans tools emerge
- You should begin using JavaBeans features immediately
  - following standard design patterns makes your code easier to maintain
  - a consistent event model also makes your code easier to maintain
  - as builder tools mature, you can continue to use your current code that follows JavaBeans guidelines

Appendix A: java.beans Package



Appendix B: Design Patterns

Simple Properties:

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

In addition, boolean properties may be retrieved by:

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

Indexed Properties:

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

You may also use this style for Indexed Properties:

```
public <PropertyElement>[] get<PropertyName>();
public void set<PropertyName>(<PropertyElement>[] a);
```

Patterns for Events:

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

The java.beans.Introspector class has a method which will decapitalize property names as follows:

“FooBah” becomes “fooBah”

“Z” becomes “z”

“URL” becomes “URL”

Appendix C: New Bean Checklist

- Must have a public, no-arg constructor
- Packaged in a JAR file, with Manifest entry specifying: Java-Bean: True
- Uses JavaBeans “design patterns”, or has an explicit BeanInfo class
- Follows the JDK 1.1 event model
- Will work in an untrusted applet
- Will work in a multi-threaded environment
- Implements Serializable or Externalizable

DigitalClock.java

```
package COM.ociweb.courses.javabeans.clock;

import java.awt.*;
import java.beans.PropertyChangeEvent;
import java.beans.PropertyChangeListener;
import java.beans.PropertyChangeSupport;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.Serializable;
import java.text.SimpleDateFormat;
import java.util.Date;
import java.util.GregorianCalendar;

/**
 * This is a simple Bean that displays a digital clock.  The clock updates
 * its display every second, and allows the user to specify colors as
 * well as the format of the text that is displayed.
 * @author Eric Burke
 */
public class DigitalClock extends Label implements Runnable, Serializable  {

    /** The JavaBeans property for the background color. */
    public static final String BACKGROUND_PROPERTY = "Background";

    /** The JavaBeans property for the foreground color. */
    public static final String FOREGROUND_PROPERTY = "Foreground";

    /** The JavaBeans property for the display format. */
    public static final String DISPLAYFORMAT_PROPERTY = "Display Format";

    /** The JavaBeans property for the font. */
    public static final String FONT_PROPERTY = "Font";

    private PropertyChangeSupport changes = new PropertyChangeSupport(this);
    private String displayFormat = "MMMM d, yyyy h:mm:ss a";

    // transient fields will not be stored when this object is serialized.
    private transient SimpleDateFormat formatter;
    private transient Thread timeThread;
```

DigitalClock.java

```
/**
 * Construct a new Digital Clock.  All Beans require a no-arg constructor.
 */
public DigitalClock() {
    init();

    // default colors
    setForeground(Color.green);
    setBackground(Color.black);
    updateDisplay();
}

/**
 * Implement the runnable interface.  The display will be updated
 * every second.
 */
public void run() {
    while(true) {
        try {
            Thread.sleep(1000);
        }
        catch(InterruptedException e) {}
        updateDisplay();
    }
}
```

DigitalClock.java

```
/**
 * Set the pattern to use for the display.  The patterns are taken from
 * the java.text.SimpleDateFormat class.
 * @param newFormat the new display format.
 */
public synchronized void setDisplayFormat(String newFormat) {
    String oldFormat = getDisplayFormat();

    try {
        formatter.applyLocalizedPattern(newFormat);
        updateDisplay();

        // make sure the bean properly updates its size
        invalidate();
        Component p = getParent();
        if (p != null) {
            p.validate();
        }
        changes.firePropertyChange(DISPLAYFORMAT_PROPERTY,
                                   oldFormat, newFormat);
    }
    catch(Exception e) {
        // revert to the original format if anything went wrong
        formatter.applyLocalizedPattern(oldFormat);
    }
}

/**
 * @return the display format.
 */
public synchronized String getDisplayFormat() {
    return formatter.toLocalizedPattern();
}
```

DigitalClock.java

```
/**
 * Set the new font to use.  This is a bound property.
 * @param newFont the new font for this bean.
 */
public synchronized void setFont(Font newFont) {
    Font oldFont = getFont();
    super.setFont(newFont);
    invalidate();
    Component p = getParent();
    if (p != null) {
        p.validate();
    }

    changes.firePropertyChange(FONT_PROPERTY,
                               oldFont, newFont);
}

/**
 * Set the background color.  This is a bound property.
 * @param newColor the new color to use.
 */
public synchronized void setBackground(Color newColor) {
    Color oldColor = getBackground();
    super.setBackground(newColor);
    changes.firePropertyChange(BACKGROUND_PROPERTY,
                               oldColor, newColor);
}

/**
 * Set the new foreground color.  This is a bound property.
 * @param newColor the new foreground color.
 */
public synchronized void setForeground(Color newColor) {
    Color oldColor = getForeground();
    super.setForeground(newColor);
    changes.firePropertyChange(FOREGROUND_PROPERTY,
                               oldColor, newColor);
}
```

DigitalClock.java

```
    /**
     * Register a PropertyChangeListener with this bean.  The listener
     * will get notified whenever a bound property is modified.
     * @param l the listener to register.
     */
    public void addPropertyChangeListener(PropertyChangeListener l) {
        changes.addPropertyChangeListener(l);
    }

    /**
     * Unregister a listener.
     * @param l the PropertyChangeListener to unregister.
     */
    public void removePropertyChangeListener(PropertyChangeListener l) {
        changes.removePropertyChangeListener(l);
    }

    /**
     * Refresh the text that is displayed.
     */
    private synchronized void updateDisplay() {
        Date curTime = GregorianCalendar.getInstance().getTime();
        setText(formatter.format(curTime));
    }

    /**
     * Initialize any transient fields in this bean.
     */
    private synchronized void init() {
        // SimpleDateFormat was not de-serializing properly, so it was
        // made into a transient field.
        formatter = new SimpleDateFormat();
        formatter.applyLocalizedPattern(displayFormat);
        timeThread = new Thread(this);
        timeThread.start();
    }

    // Read this bean from persistent storage.
    private void readObject(ObjectInputStream in)
    throws IOException, ClassNotFoundException {
        in.defaultReadObject();  // deserialize component in the usual way
        init();                  // initialize transient fields
    }
}
```

DigitalClockBeanInfo.java

```
package COM.ociweb.courses.javabeans.clock;

import java.awt.Image;
import java.beans.BeanDescriptor;
import java.beans.BeanInfo;
import java.beans.IntrospectionException;
import java.beans.PropertyDescriptor;
import java.beans.SimpleBeanInfo;

/**
 * This class provides explicit information about the DigitalClock Bean.
 * @author Eric Burke
 */
public class DigitalClockBeanInfo extends SimpleBeanInfo {

    /**
     * Provide an Icon that builder tools can use for palettes or tool bars.
     * @param iconKind an enumeration from java.beans.BeanInfo.
     * @return the appropriate image, or null if iconKind is invalid.
     */
    public Image getIcon(int iconKind) {
        if (iconKind == BeanInfo.ICON_COLOR_16x16) {
            Image img = loadImage("DigitalClockIconColor16.gif");
            return img;
        }
        if (iconKind == BeanInfo.ICON_COLOR_32x32) {
            Image img = loadImage("DigitalClockIconColor32.gif");
            return img;
        }
        if (iconKind == BeanInfo.ICON_MONO_16x16) {
            Image img = loadImage("DigitalClockIconMono16.gif");
            return img;
        }
        if (iconKind == BeanInfo.ICON_MONO_32x32) {
            Image img = loadImage("DigitalClockIconMono32.gif");
            return img;
        }
        return null;
    }
```

DigitalClockBeanInfo.java

```
/**
 * This method returns a BeanDescriptor object.  The BeanDescriptor
 * is used to obtain a meaningful name for this bean to be used in
 * builder tools.  Although the BeanDescriptor can also be used to
 * provide a customizer, the DigitalClock bean has no customizer.
 * @return the BeanDescriptor object for DigitalClock.
 */
public BeanDescriptor getBeanDescriptor() {
    BeanDescriptor bd = new BeanDescriptor(DigitalClock.class);
    bd.setDisplayName("Digital Clock");
    return bd;
}

/**
 * This method returns explicit information about the properties
 * supported in this bean.
 * @return an array of objects which describe this bean's properties.
 */
public PropertyDescriptor[] getPropertyDescriptors() {
    try {
        PropertyDescriptor displayFormat = new PropertyDescriptor(
            DigitalClock.DISPLAYFORMAT_PROPERTY,
            DigitalClock.class,
            "getDisplayFormat",
            "setDisplayFormat");

        PropertyDescriptor foreground = new PropertyDescriptor(
            DigitalClock.FOREGROUND_PROPERTY,
            DigitalClock.class,
            "getForeground",
            "setForeground");

        PropertyDescriptor background = new PropertyDescriptor(
            DigitalClock.BACKGROUND_PROPERTY,
            DigitalClock.class,
            "getBackground",
            "setBackground");
```

DigitalClockBeanInfo.java

```
            PropertyDescriptor font = new PropertyDescriptor(
                DigitalClock.FONT_PROPERTY,
                DigitalClock.class,
                "getFont",
                "setFont");

            displayFormat.setBound(true);
            foreground.setBound(true);
            background.setBound(true);
            font.setBound(true);

            PropertyDescriptor[] pda = { displayFormat,
                                         foreground,
                                         background,
                                         font };

            return pda;
        }
        catch(IntrospectionException e) {
            return null;
        }
    }
}
```