Java Beans

By:

Mr.Ganesh Kr. Yadav

Department of Computer Science & Engineering,
ABESIT College of Engineering, Ghaziabad.

What is Java Bean?

- A Java Bean is a software component. This component has been designed to be reusable in a variety of different environments.
- A Java Bean is a reusable software component that can be manipulated visually in a builder tool (Visual Age, Delphi, Visual Basic, PowerBuilder etc.)
- Developers can use software components written by others without having to understand their inner workings.

  To understand why software components are useful, think of a worker assembling a car. Instead of building a radio from scratch, for example, he simply obtains a radio and hooks it up with the rest of the car.
- There is no restriction on the capability of a Bean. It may perform a simple function, such as spelling check of a document, or a complex function such as forecasting the performance of a stock portfolio.
- A Bean may be **visible** to an end user (ex. A button)
- A Bean may be **invisible** to a user (ex. Software to decode a stream of multimedia information in a real time)
- Beans are dynamic in such a way that they can be changed or customized.
- The classes that define the beans, referred to as ***JavaBeans components*** **or simply** ***components***

Advantages of Java Beans

- Java beans are reusable software components
- A Bean obtains all the benefits of Java's write-once, run-anywhere paradigm.
- The properties, events, and methods of a Bean that are exposed to another application, such as a builder tool, can be controlled.
- The configuration settings of a Bean can be saved in persistent storage and restored at a later time.
- We can use bean to register the receive events from other objects and generate events that are sent to other objects
- Auxiliary software can be provided to help a person to configure a Bean. This software is only needed when the design-time parameters for that component are being set. In runtime environment, it does not need to be included.
- One of the major applications of JavaBeans is for rapid application development, or RAD, a powerful feature used in software-development tools for visual programming. RAD allows you to select a component and drop it in a desired location. RAD in Java builder tools is completely based on JavaBeans.

Introspection

- Introspection is the process by which a builder tool analyzes how a bean works and also differentiates beans from the typical java classes.
- Introspection as the process of analyzing a bean to determine its capabilities like which properties, methods and events a bean supports.
- It is an essential feature of the JavBeans API since it allows a builder tool to obtain the information about a component.
- A builder tool discover a bean's features by a process known as *introspection*.

➢ **Beans support introspection in two ways:**

- **By adhering to specific rules (*design patterns*) when naming properties, methods and events**
- **By explicitly providing property, method and event information within a *Bean Information* class**

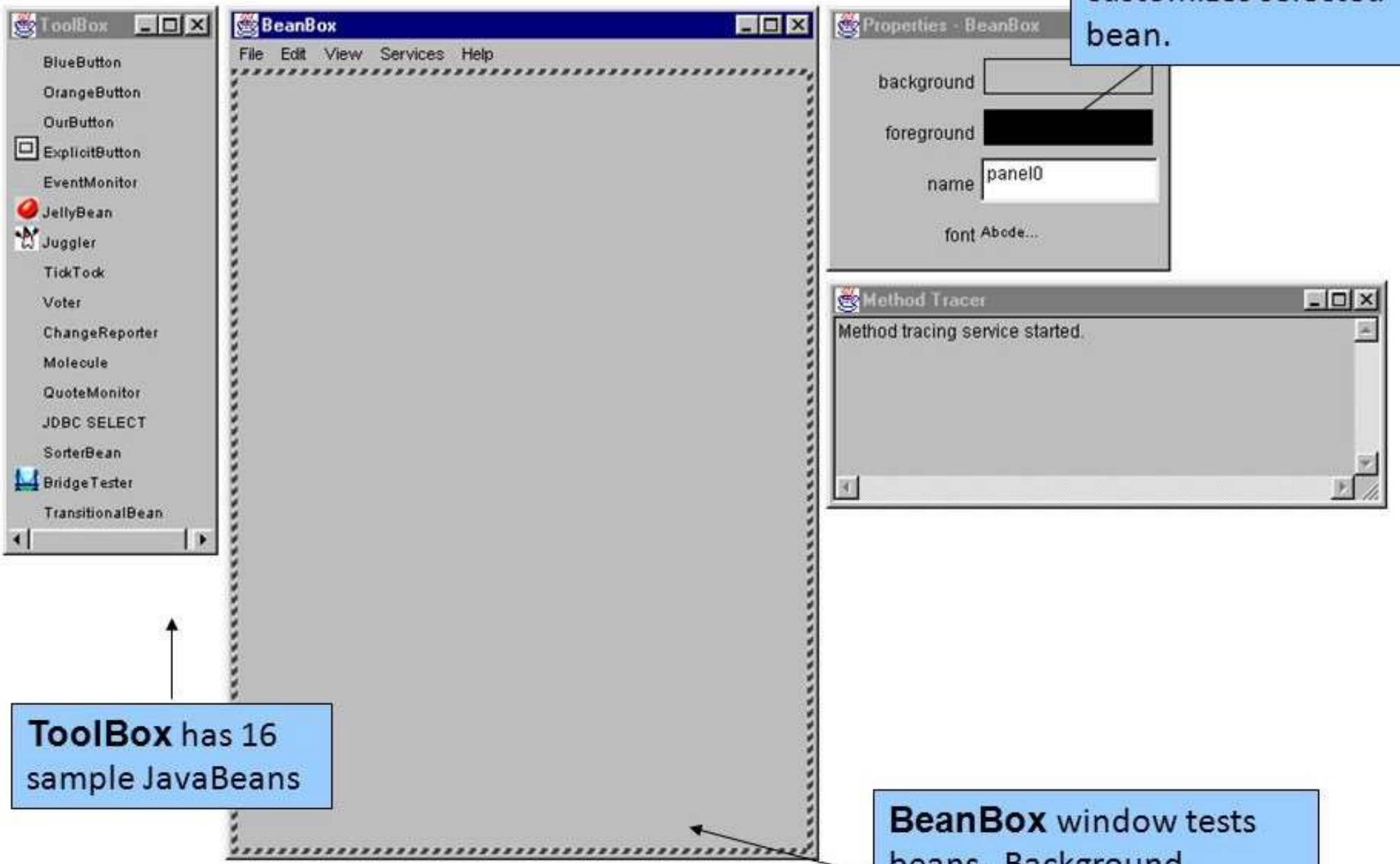
Customizers

- Customization is nothing but the properties (appearance and behaviour of a bean)can be changed at design time.
- Two ways to customize a bean:
  - using a property editor
    - each bean has its own property editor
  - using bean customizers
    - gives you complete GUI control over bean customization
- A bean developer provides a customizer to help another developer to configure the bean
- A customizer can provide a step by step guide that must be followed to use the component in a specific context.
- Online documentation can also be provided.
- Through the design mode of a builder tool, you can use the property sheet or bean customizer to customize the bean and then save (persist) your customized beans.

Designing Programs using Java Beans

- The Beans Development Kit (BDK) is a pure Java application which provides support for the JavaBeans APIs, a test container (the "BeanBox" to test Bean behavior)
- The BeanBox is a sample container for testing Beans. The BeanBox handles visible Beans, those Beans that have a visual element, and invisible Beans, those Beans that are purely computational objects.
- After downloading the BDK, to start or open the BeanBox:
  - Run the batch file run.bat (Windows)
  - Run the file run.sh (Unix)
- When you start the BeanBox, **a ToolBox** of sample Beans is displayed.
- When you add beans to an application they will appear in the toolbox of the **beanbox window.**
- **Properties sheet:** displays the properties for the Bean currently selected within the BeanBox window and it can be used to set the properties that have designed into a bean.
- We can use the **Method Tracer window** to handle the method execution.

BeanBox Overview



Properties customizes selected bean.

ToolBox has 16 sample JavaBeans

BeanBox window tests beans. Background currently selected (dashed box).

Develop a New Bean

- Create a directory for the new bean
- Create the java bean source file(s)
- Compile the source file(s)
- Create a manifest file
- Generate a JAR file
- Install in jars directory
- Start BDK
- Test

Creating a Java Bean

Example: This bean will just draw itself in red, and when you click it, it will display a count of the number of time it has been clicked. We can place this bean in the BDK's demo directory, so we create a directory named bean and store the class files for this bean in that directory.

In this bean class we can use Canvas to draw the bean itself. And add a mouse listener to the canvas to record mouse clicks and set the size of the canvas

```
package sunw.demo.bean;
import java.awt.*;
import java.awt.event.*;
public class bean extends Canvas {
    int count;
    public  bean()
    {       addMouseListener(new  MouseAdapter()
                                {   public void mousePressed(MouseEvent me)
                                    {
                                        clicked();
                                    }
                                });
```

```
            count = 0;
            setSize(200,100);
        }
public void clicked()
{
    count++;
    repaint();
}
public void paint(Graphics g)
{
   g.setColor(Color.RED);
   g.fillRect(0,0,20,30);
   g.drawString("Click Count= "+count,50,50);
}
```

- Now this bean class have to put it into a JAR file and label it in that JAR file as a bean

- Compile the bean.java file ; then bean.class file will be generated.
- After creating the bean, create the manifest file for the bean class

Creating a Bean Manifest File

- We use a manifest to indicate which classes are beans.
- To indicate that a class in a JAR file is a Java Bean, you have to set its Java-Bean attribute to True.

bean.mft file

Name: sunw/demo/bean/bean.class

Java-Bean: True

Attributes:

- **`Name:`** name of the bean class (full package and class name)
- **`Java-Bean:`** **`true`** – indicate that a class in a JAR file is a Java Bean

Creating a Bean jar file:

- To use a bean, you have to store the class file(s) and manifest file in a JAR file.
- We can create a JAR file for this bean, i.e. **bean.jar** in the **demo\jars** directory located within the beans directory, where the beanbox will look for it.
- Make sure you are in the demo directory and use the jar tool like

jar cfm ..\jars\bean.jar bean.mft sunw\demo\bean\bean.class

  - `jar` utility at command line
  - Options
    - `c` – creating new JAR file
    - `f` – indicates archive filename
    - `m` – specify the manifest file
- Bean is compressed and saved in the format of jar file which contains manifest file, class files, gif files, and other information of customization files

Using the New Bean:

- After developing a new JavaBean and installing it in the demo\jars directory, we can open the beanbox to see this bean listed in the toolbox
- When you draw the bean from the toolbox, the bean appears in the BeanBox window

Adding Controls to a Bean:

- We can add a button to a bean and display the number of times it has been clicked.
- First base your bean on a class that has a container, such as the Panel class
- This button.java file creates the panel, sets it size, and adds a button to it.

```
package sunw.demo.button;
Import java.awt.*;
Import java.awt.event.*;
public class button extends Panel implements ActionListener
{
    int count;
    Button  button1;
    public button()
    {
        count=0;
        setSize(200,100);
```

```
        button1=new Button("Click me");
        button1.addActionListener(this);
        add(button1);
    }
    public void actionPerformed(ActionEvent  e)
{
        count++;
       repaint();
    }
    public void paint(Graphics   g)
    {
       g.drawString("Click count= "+count,50,50);
    }
    }
```

Giving a Bean Properties

- A bean *property* is a named attribute of a bean that can affect its **behavior or appearance.**
- Examples of bean properties include color, label, font, font size, and display size.
- Bean's properties (i.e appearance and behaviour) can be changed at design time.
- Bean properties are private values and can be accessed through getter and setter methods
- The names of these methods follow specific rules called ***design patterns***.
- These design pattern-based method names allow builder tools such as the NetBeans GUI Builder, to provide the following features:
- A builder tool can:
  - discover a bean's properties
  - determine the properties' read/write attribute
  - locate an appropriate "property editor" for each type
  - display the properties (in a sheet)
  - alter the properties at design-time

- We should inform to the Java framework about the properties of your beans by **implementing the BeanInfo interface**.
- Most beans don't implement the BeanInfo interface directly. Instead, they extend the **SimpleBeanInfo class**, which implements BeanInfo interface.
- To actually describe a property, we can use the **PropertyDescriptor class**, which in turn is derived from the **FeatureDescriptor class**.

Let us take an example that implements a property in a Java bean

- We will add a property named **filled** to the click-counting operation.
- This property is a boolean property that, when True, makes sure the bean will be filled in with color.
- To keep track of the new filled property, we will add a **private boolean variable** of that name to the **bean2 class**
- We initialize this property to False when the bean is created
- When we implement a property, java will look for two methods: **getPropertyName and setPropertyName,** where PropertyName is the name of the property.
- The get method returns the current value of the property, and the set method takes an argument of that type

```
package sunw.demo.bean2;
import java.awt.*;
import java.awt.event.*;
public class  bean2 extends Canvas{
    private boolean filled ;
    int count;
    public  bean2()
    {       addMouseListener(new  MouseAdapter()
                                {   public void mousePressed(MouseEvent me)
                                    {
                                        clicked();
                                    }
                                });
            count = 0;
            filled = false;
            setSize(200,100);
        }
    public void clicked()
    {
      count++;
      repaint();
    }
```

```
public boolean getFilled()
{
    return filled;
}
public void setFilled(boolean  flag)
{
     this.filled = flag;
     repaint();
}
public void paint(Graphics g)
{
   if (filled)
   {
      g.setColor(Color.RED);
      g.fillRect(20,5,20,30);
   }
   g.setColor(Color.WHITE);
   g.drawString("Click Count= "+count,50,50);
}
```

Implementing a BeanInfo class for the property:

- Using **BeanInfo class** you can expose the features of your bean to java framework
- In BeanInfo class, the bean developer has to give the description about each property by using this PropertyDescriptor class.
- The bean info class should extend the **SimpleBeanInfo** class which implements **BeanInfo** interface
- Now we have to create a new class, **bean2BeanInfo** which will return information about this new bean property
- In this BeanInfo class we should implement the **getPropertyDescriptors** methods, which returns an array of **PropertyDescriptor** objects.
- Each PropertyDescriptor object holds the name of a property and point to the class that supports that property.

```
package sunw.demo.bean2;
import java.beans.*;
public class bean2BeanInfo extends SimpleBeanInfo
{
 public  PropertyDescriptor[]  getPropertyDescriptors()
 {
   try
   {   PropertyDescriptor  filled  = new  PropertyDescriptor("filled" ,
                                                          bean2.class);
       PropertyDescriptor   pd[]  = { filled };
       return  pd;
    }
    catch(Exception e)  { }
    return null;
}
```

- After compiling this new class(**bean2BeanInfo.java**), we place **bean2BeanInfo.class file** in the directory **sunw\demo\bean2** along with the classes created when we compiled **bean2.java**
- Create a new manifest file that includes the bean2BeanInfo class

bean2.mft file:

Name: sunw/demo/bean2/bean2BeanInfo.class

Name: sunw/demo/bean2/bean2.class

Java-bean : True

- Place this new manifest file in the demo directory.
- Create a new **bean2.jar** file and install it:

bean2.jar file:

C:\...\demo > jar cfm ...\jars\bean2.jar bean2.mft sunw\demo\bean2*.class

- Now when we run the beanbox and add a new bean2 bean to the beanbox, the new filled property will appear in the properties window.
- Setting filled to True causes the bean to be filled with color.

Constructors of the PropertyDescriptor Class:

- **PropertyDescriptor(String propertyName, Class beanClass)**

  Constructs a property descriptor for a property

- **PropertyDescriptor(String propertyName, Class beanClass, Sring getterName, String setterName)**

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

- **PropertyDescriptor(String propertyName, Method getter ,Method setter)**

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

Methods of the PropertyDescriptor Class:

- **Class getPropertyEditorClass()** - Returns any explicit PropertyEditor class that has been registered for this property
- **Class getPropertyType()** - Returns the Class object for the property

- **Method getReadMethod()** – Returns the method that should be used to read the property value
- **Method getWriteMethod()** – Returns the method that should be used to write the property value
- **boolean isBound()** - Returns true if this is a bound property
- **boolean isContrained()** - Returns true if this is a constrained property
- **void setBound(boolean bound)** - Sets the bound property of this object. Updates to bound properties cause a PropertyChange event to be fired when the property is changed.
- **void setConstrained(boolean constrained)** – Sets the constrained property of this object.
- **void setReadMethod(Method getter)** – Sets the method that is used to read the property value
- **void setWriteMethod(Method setter)** – Sets the method that is used to write the property value
- **void setPropertyEditorClass(Class propertyEditorClass)** – Returns the class for the desired PropertyEditor. Property editors are found by using the property editor manager

Design Patterns for Properties

- The behavior and appearance of that component is determined by the values assigned to the properties.
- A property is set by a setter() method and a property is obtained by a getter() method
- There are two types of properties
- These are **simple and indexed**

Bean properties can also be classified as follows:

- Write Only – A bean property that can be changed
- Read Only – A bean property that cannot be changed.
- Read/Write - A bean property can be read and write
- Hidden – A bean property that can be changed. However, these properties are not disclosed with the BeanInfo class

Simple Properties

- A simple property represents a single value and can be defined with a pair of get/set methods.
- It can be identified by the following design patterns, where N is the name of the property and T is its type.

public T getN()

public void setN(T arg)

- A read/write property has both the getN() and setN() methods.
- A read-only property has only a getN() method,
- A write-only property has only a setN() method.

The following are 3 read/write simple properties along with their getter() and setter() mehods:

```
private double depth,height,width;
public double getDepth()  {   return depth;  }
public void setDepth(double  dp) { depth = dp; }
public double getHeight()  {   return height;  }
public void setHeight(double  ht) { height = ht; }
public double getWidth()  {   return width;  }
public void setWidth(double  wd) { width = wd; }
```

Adding a Color Property to SimpleBean

```
public  class  SimpleBean  extends Canvas implements Serializable
{
    private  Color  color = Color.green;   // Property name is clr

        public   SimpleBean()   // default constructor
        {
            setSize(60,40);
            setBackground(Color.red);
        }
        public  Color getColor()          //   get property
        {
           return color;
        }
```

```
    public void setColor(Color c)   //  set  property
    {
        color = c;
        repaint();
    }
    public void paint(Graphics g){
       g.setColor(color);
       g.fillRect(20,5,20,30);
    }
}
```

- When you execute the bean, you will get following results:

❑ SimpleBean will be displayed with a green centered rectangle.

❑ The Properties window contains a new clr property

Indexed Properties:

- Indexed properties consist of multiple values.
- Property element get/set methods take an integer index as a parameter.
- The property may also support getting and setting the entire array at once.
- Multiple values are passed to and retrieved from the setter and getter methods respectively in case of the indexed properties.
- It can be identified by the following design pattern, N is the name of the property and T is its type

```
public void setN(int index, T value);
public T getN(int index);
public T[] getN();
public void setN(T values[]);
```

```
private double data[];     //  data is an indexed property
public double getData(int  index)  //get one element of array
{
    return data[index];
}
public void setData(int index, double   x)   //set one element of array
{
   data[index]=x;
}
public  double []   getData()    //   get entire array
{
   return data;
}
public void setData(double[]  x)   // set entire array
{
   data=x;    }
```

Design Patterns for Events

- The Beans can perform various functions, such as it can generate events and send them to other objects.
- These can be identified by the following design patterns, where T is the type of the event:

public void addTListener(TListener eventListener);

public void removeTListener(TListener eventListener);

- To add or remove a listener for the specified event, these methods are used.
- For example, an event interface type is **TemperatureListene**r, a Bean that monitors temperature might supply the following methods:

```
public void addTemperatureListener(TemperatureListener   tl)
{ . . . }
public void removeTemperatureListener(TemperatureListener  tl)
{ . . . }
```

Using the BeanInfo Interface

- The role of BeanInfo interface is to enable you to explicitly control what information is available.
- To create a BeanInfo class , you should extends **SimpleBeanInfo** class which implements **BeanInfo interface**
- The BeanInfo interface defines several methods. Like

  PropertyDescriptor[] getPropertyDescriptors()

  EventSetDescriptor[] getEventSetDescriptors()

  MethodDescriptor[] getMethodDescriptors()
- Here, these array of objects provide information about the properties, events, and methods of a Bean
- By implementing these methods, a developer can designate exactly what is presented to a user
- At the time of creating a class that implements BeanInfo, you have to remember that you must call that class **bnameBeanInfo**, where **bname** is the name of the Bean.
- For example, if the Bean is called as **MyBean** then the information class must be called **MyBeanBeanInfo**

Feature Descriptors

- BeanInfo classes contain descriptors which describe the target Bean's features.
- **FeatureDescriptor**- It is the base class for the other descriptor classes. It declares the aspects that are common to all descriptor types.
- **PropertyDescriptor** – It describes the target Bean's properties
- **IndexedPropertyDescriptor** – It describes the target Bean's indexed properties
- **EventSetDescriptor**- It describe the events the target Bean fires
- **MethodDescriptor**- It describes the target Bean's methods.
- **ParameterDescriptor** – It describes the method parameters

Creating a BeanInfo Class

- To create a **BeaInfo class** , first name your BeanInfo class .
- According to naming convention, you must add **“BeanInfo”** to the target class name.
- If the target class name is **ExplicitButton** then its bean information class name should be **ExplicitButtonBeanInfo.**
- By extending **SimpleBeanInfo** class you can override only those methods which returns the properties, methods and events

```
public class ExplicitButtonBeanInfo extends SimpleBeanInfo
{
   public ProertyDescriptor[]  getPropertyDescriptors()
  {
 PropertyDescriptor  background = new PropertyDescriptor("background",
                                                          "ExplicitButton.class");
PropertyDescriptor  foreground = new PropertyDescriptor("foreground",
                                                          "ExplicitButton.class");
PropertyDescriptor  font = new PropertyDescriptor("font", "ExplicitButton.class");
PropertyDescriptor  label= new PropertyDescriptor("label",
                                                          "ExplicitButton.class");
PropertyDescriptor   pd[] = {background,foreground,font,label};
return pd;
}
}
```

Giving a Bean Methods

- We can declare a method in a JavaBean, which can be called by other beans.
- Any public bean method is accessible from other beans.
- We can describe the methods of JavaBeans to the Java framework by using the **MethodDescriptor** class.

Constructor Summary

MethodDescriptor(Method method)

- Constructs a MethodDescriptor from a Method.

MethodDescriptor(Method method, **ParameterDescriptor**[] parameterDescriptors)

- Constructs a MethodDescriptor from a Method providing descriptive information for each of the method's parameters.

Method Summary:

Method getMethod() - Gets the method that this MethodDescriptor encapsualtes.

ParameterDescriptor[] getParameterDescriptors() - Returns the ParameterDescriptor for each of the parameters of this MethodDescriptor's methods.

- This Bean3 will count the number of times it has been clicked and will also support a method named increment that, when invoked, will increment the click count.

```
public class  bean3 extends Canvas {
    int count;
    public  bean3()
    {       addMouseListener(new  MouseAdapter()
                        {   public void mousePressed(MouseEvent me)
                            {
                                clicked();
                            }
                        });
```

```
            count = 0;
             setSize(200,100);
        }
public void clicked()
    {
      count++;
      repaint();
    }
 public void increment()    /*public bean method is
                            accessible from other beans  */
    {
      count++;
      repaint();
    }
public void paint(Graphics g)
{
   g.setColor(Color.RED);
   g.fillRect(0,0,20,30);
   g.drawString("Click Count= "+count,50,50);
}
```

- After creating bean3 and adding it to the beanbox, we can connect other beans to the increment methods.
- We can connect a button to that method, and each time the button is clicked, the click count in the bean3 is incremented and displayed.
- In order to make available the methods defined in one bean to another bean, we have to create a BeanInfo class and implement the method called **getMethodDescriptors()**

```
public MethodDescriptor[]  getMethodDescriptors()
{
   MethodDescriptor   m1 = new MethodDescriptor("increment");
   MethodDescriptor  md[] = {m1};
   return   md;
 }
```

Bound Properties:

- Bound properties generates an event when their values change.
- This event is of type **PropertyChangeEvent** and is sent to all registered event listeners of this type.
- To make a property a bound property, use the **setBound** method like

  PropertyName.setBound(true)

  For example **filled.setBound(true);**
- When bound property changes, an event is of type **PropertyChangeEvent**, is generated and a notification is sent to interested listeners.
- There is a standard listener class for this kind of event. Listeners need to implement this interface **`PropertyChangeListener`**

  It has one method:

  `public void propertyChange(PropertyChangeEvent)`
- A class that handles this event must implement the **PropertyChangeListene**r interface

Implement Bound Property in a Bean

- Declare and Instantiate a **PropertyChangeSupport** object that provides the bulk of bound property's functionality,

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

- Implement registration and unregistration methods . The BeanBox will call these methods when a connection is made.

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

- Send change event to listeners when property is changed. i.e each bound property must invoke the **firePropertyChange()** method from its set() method:
- PropertyChangeSupport object handles the notification of all registered target.
- The method **firePropertyChange()** must provide the property name, as well as the old and new values

```
public void setX(int new)
{
        int   old = x;
        x = new;
        changes.firePropertyChange("x", old, new);
}
```

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

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

Constrained Properties:

- An object with constrained properties allows other objects to veto a constrained property value change.
- A bean property for which a change to the property results in validation by another bean. The other bean may reject the change if it is not appropriate.
- Constrained property listeners can veto a change by throwing a **PropertyVetoException.**
- It generates an event called **PropertyChangeEvent** when an attempt is made to change its value
- This event is sent to objects that previously registered an interest in receiving an such notification
- Those objects have the ability to veto the proposed change
- This allows a bean to operate differently according to the runtime environment

Bean with constrained property must

1. Allow **VetoableChangeListener** object to register and unregister its interest in receiving notifications
2. Fire property change at those registered listeners. The event is fired before the actual property change takes place

Implementation of Constrained Property in a Bean

1. To support constrained properties the Bean class must instantiate the a **VetoableChangeSupport** object

```
private VetoableChangeSupport vetos=new
                                    VetoableChangeSupport(this);
```

2. Define registration methods to add and remove vetoers.

```
public void addVetoableChangeListener(VetoableChangeListener v)
{
    vetos.addVetoableChangeListener(v);
}
public void removeVetoableChangeListener(VetoableChangeListener v)
{
    vetos.removeVetoableChangeListener(v);
}
```

3. Write a property's setter method to fire a property change event and setter method throws a **PropertyVetoException**. In this method change the property and send change event to listeners.

```
public void setX(int new) throws PropertyVetoException
    {
        int old=X;
        vetos.fireVetoableChange("X", old, new);
        //if no veto  is there
        X=new;
        changes.firePropertyChange("X", old, new);
    }
```

- In general, constrained properties should also be bound. The source should notify any registered vetoableChange listeners that a vetoableChange has been proposed. This object either accepts or rejects the proposed change.
- If the change is acceptable, the source notifies any registered propertyChange listeners that the change has completed

4. A veto-interested target object must implement the **vetoableChange()** method:

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

EventSetDescriptor Class

- An **EventSetDescriptor** describes a group of events that a given Java bean fires.
- The given groups 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.

Constructor:

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

sourceClass - The class firing the event.

eventSetName - The programmatic name of the event.

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.

Throws: IntrospectionException - if an exception occurs during introspection

Code Snippet:

```
public EventSetDescriptor[] getEventSetDescriptors()
            //describe the events supported by the bean
{
  try
  {
   EventSetDescriptor changed=new EventSetDescriptor(
                              beanClass,"propertyChange“,
        PropertyChangeListener.class,"propertyChange");

   changed.setDisplayName(“FieldPanel value changed“);
               // set the title of event in builder tool

   EventSetDescriptor[]  ed = { changed };
   return ed;
}
catch (IntrospectionException e)
 { }
```

Persistence

- The ability to store state of a component in persistent storage is called ***persistence*.**
- A bean need to be implement the interface **java.io.Serializable** to support the persistence
- The Serializable interface do not prescribe the implementation of any methods, but is an approval, that the bean may be saved in persistent storage as in a file or database.
- In doing so, the bean can be restored after an application was shut down or transferred across networks.
- To make fields in a Bean class persistent, simply define the class as implementing java.io.Serializable.

```
public class Button implements Serializable
{ ...../////....}
```

- You can prevent selected fields from being saved by marking them transient or static; since transient and static variables are not saved.
- Alternatively, a component can be stored in a customized manner (e.g. in xml format) by implementing the **Externalizable** interface.

The Java Beans API (java.beans package)

- The JavaBeans API contains classes and interfaces that enable a Java developer to work with beans in a Java program.
- The classes, interfaces and methods of the JavaBeans API are provided in the **java.beans** package.
- With the JavaBeans API you can create reusable, platform-independent components. Using JavaBeans-compliant application builder tools, you can combine these components into applets, applications, or composite components.

[Note: Sun NetBeans, BDK, Visual Café, JBuilder, Visual Age are the bean builder tools]

List of Interfaces in java.beans package

- BeanInfo Any class that implements this BeanInfo interface provides explicit information about the methods, properties, events, etc, of their bean.
- Customizer A customizer class provides a complete custom GUI for customizing a target Java Bean.
- ExceptionListener An ExceptionListener is notified of internal exceptions.
- PropertyChangeListener A "PropertyChange" event gets fired whenever a bean changes a "bound" property.
- PropertyEditor This interface provides the methods that is used to edit the property
- VetoableChangeListener A VetoableChange event gets fired whenever a bean changes a "constrained" property.
- Visibility This interface determines that bean needs GUI or not and whether GUI is available.
- AppletInitializer This interface is used to initialize the Applet with java Bean
- DesignMode This interface is used to define the notation of design time as a mode in which JavaBean instances should function

List of Classes in java.beans package

- BeanDescriptor A BeanDescriptor provides the complete information aout the bean
- Beans This class provides some general purpose beans basic methods.
- EventHandler The EventHandler class is used by developers to make connections between user interface bean(source) and an appication logic bean(target)
- EventSetDescriptor An EventSetDescriptor describes a group of events that a given Java bean fires.
- FeatureDescriptor It is the baseclass for PropertyDescriptor, EventSetDescriptor, and MethodDescriptor, etc.
- IndexedPropertyChangeEvent It gets delivered whenever a component of JavaBeans hanges a bound indexed property.
- IndexedPropertyDescriptor It describes a property that acts like an array having index read or write method of the array.
- Introspector It class provides a standard way for tools to learn about the properties, events, and methods supported by a target Java Bean.

- MethodDescriptor A MethodDescriptor describes a particular method that a Java Bean supports for external access from other components.
- ParameterDescriptor This class allows bean implementors to provide additional information on each of their parameters
- PropertyChangeEvent A "PropertyChange" event gets delivered whenever a bean changes a "bound" or "constrained" property.
- PropertyChangeSupport This is a utility class that can be used by beans that support bound properties.
- PropertyDescriptor A PropertyDescriptor describes one property that a Java Bean exports via a pair of accessor methods.
- PropertyEditorManager The PropertyEditorManager can be used to locate a property editor for any given type name.
- PropertyEditorSupport This is a support class which helps to build property editors.
- SimpleBeanInfo This is a support class to make it easier for people to provide BeanInfo classes.
- VetoableChangeSupport This is a utility class that can be used by beans that support constrained properties.

Giving a Bean an Icon

- You can add your own icons to your beans by adding a **getIcon** method to the **BeanInfo class.**
- Implement this method and handle all possibilities like monochrome or color icons of either 16X16 or 32X32

```
pubic  Image getIcon(int   iconkind)
{
    if (iconkind ==  BeanInfo.ICON_MONO_16X16 || iconkind ==
                                                BeanInfo.ICON_COLOR_16X16)
   {
        Image   image =  loadImage("Icon16.gif");
        return image;
   }
   if (iconkind ==  BeanInfo.ICON_MONO_32X32 || iconkind ==
                                                    BeanInfo.ICON_COLOR_32X32)
  {
     Image   image =  loadImage("Icon32.gif");
     return image;
  }
      return null;
}
```

Questions ?