Unit – 5 Java Beans Components

1) Introduction to Beans ?

A bean is a reusable software component based on sun's JavaBeans specification that can be manipulated in a builder tool.

Once a bean is implemented, it can be use in a builder environment such as NetBeans or JBuilder to produce GUI applications more efficiently.

The JavaBeans technology was invented to make Java technology competitive.It enables vendors to create VB style environments for developing Java user Interfaces with a minimum of programming.

VB is one of the most successful examples of reusable object technology.

Java Beans

- can be defined as an independent software component that is developed using Java. The main usage of Javabeans is its reusability.
- A Java Beans have three fundamental components:

a) Event: - can be defined as any change in the state of a bean or in its environment.

b) Properties: - defined as private members, data that can be accessed as & when required

c) Methods: - typically in Javabeans are required to get/access or set/specify property of a beans and also we may code methods to perform another operations/tasks.

Functional Characteristics of Bean: -

1. *Communication:* One bean can communicate with other bean and application in that it is build. By message passing to method, it do communication.
2. *Introspection:* can be defined as the ways whereas we observe a bean to analyze bean events, properties or methods.
3. *Persistence:* storing states of a bean or the bean itself over a permanent storage. So that in future if we need the same then it can be restored.
4. *Customizers:* - modifying the appearance and behavior at the time of building, and run time.

To implement and execute JavaBean need BDK (Bean Development Kit)

- Steps to start beanbox & create an example of Juggler bean: -
  C://BDK1.1/beanbox>run.bat

- When start beanbox, 4 windows are open

1. **BeanBox:-** work space when we put tools from toolbox or we may load it from any directory when you have saved it.
2. **Properties:-** window enables us to edit any property of selected control.
3. **Method Tracer:-** specify/display the properties that is being performed.
4. **ToolBox: -** contains loaded beans.

2) The Bean-Writing Process –

The simplest kind of bean is really nothing more than a java class that follows Builder tools use this standard naming conventions for its methods. Properties are conceptually at a higher level than instance fields –they are features of the interface, whereas instance fields belong to the implementation of the class.

Real - world Beans are much more elaborate & tedious to code, because of two reasons:

1)Beans must be usable by less-than-expert programmers. U need to expose lots of properties so that your users can access most of the functionality of your bean with a design tool & without programming.

2) The same bean must be usable in a wide variety of contexts. Both the behavior & appearance of your bean is customizable.

A good example of a bean with rich behavior is CalendarBean by Kai Todter.

This bean gives user a convenient way of entering dates, simply by locating them in a calendar display.

By using a bean such as this one, u can take advantage of the work of others, simply by dropping the bean into a builder tool.

- In Java beans program, we may define property & these property can be edited as & when required.
- To declare property(s), we declare private members data/private member variables in a Java bean program.
- To edit these properties Java Beans provides two methods: -
    1. *Accessor Method:* - this method enables us to retrieve Java bean properties. Here getProperty() method is used.
    2. *Mutator Method:* - is used to specify or set a property in Java bean program, use setProperty().

Bean writing is a process to defining the properties, registering with events and implementing the methods. It is the construction state of the bean.

Following steps must follow to create a new bean: -

1. *Create a directory for the new bean*
2. *Create the Java source file*
3. *Compile the source file: - As usual by using Javac compiler: - javac FileName.java after compilation FileName.class file shall get created*
4. *Now create a manifest file or mft file as: -open notepad→ Type the folowing :*
   Name: FileName.class
   Java-Bean: True
   After Java-Bean: True, press Enter Key (is compulsary).
   Save it under C://jdk/bin/Directory for new bean
5. *Now generate a JAR file as : - C://jdk/bin/Dir>jar cfm FileName.jar FIleName.mft *.class*
   ///.class is used because a bean when is compiled creates more than one class*
6. *Start the BDK as: - C://BDK/beanbox> run.bat*
7. *From beanbox, goto file menu option → load jar and then select: - C://jdk/bin/Dir>FileName.jar*
8. *You will find "FileName" in ToolBox, drag & drop it over beanbox frame*
9. *Change its logo property from "property" box*
10. *Now drag an OurButton object from toolbox to beanbox & then change its title as "Click Here", select it, and then bind action Performed →Events→Action→ActionPerformed→ValidatePin)→OK (Edit*
11. *Enter any Pin No. in the text field of Pin No & then click button "Click Here".*

3) Using Beans to Build an Application -

Builder environments aim to reduce the amount of drudgery that is involved in wiring together components into an application.
Each builder environment uses its own strategies to ease the programmer's life. The NetBeans is better integrated development environment because it is a fairly typical programming environment & it is freely available also.

Packaging Beans in JAR Files –

To make any bean usable in a builder tool, package into a JAR file all class files that are used by the bean code.
A JAR file for a bean needs a manifest file that specifies which class files in the archive are beans & should be included in the ToolBox.
If your bean contains multiple class files, just mention in the manifest those class files that are beans & that you want to have displayed in the toolBox.

To make the JAR file, follow these steps :

1) *Edit the manifest file.*
2) *Gather all needed class files in a directory.*
3) *Run the jar tool as follows:*
   jar cvfm JarFile ManifestFile ClassFiles

Ex:

jar cvfm ImageViewerBean.jar ImageViewerbean.mf
com/horstmann/corejava/.class*

You can also add other items such as GIF files for icons,to the JAR file.
Builder Environments have a mechanism for adding new beans, typically
By loading JAR files. Here is what you do to import beans into NetBeans. Compile the ImageViewerBean & FileNameBean classes & package them into JAR files. Then start NetBeans & follow these steps :

1) *Select Tools --> Palette Manager from the menu.*
2) *Click the Add from JAR button.*

In the file dialog box, move to the ImageViewerBean directory & select ImageViewerBean.jar

Now a dialog box pops up that lists all the beans that were founding the JAR file.Select ImageViewerBean.

Finally, you asked into which palette you want to place the beans. Select Beans.

Have a look at the Beans palette.It now contains an icon representing the new bean.

A simple Bean Programme

```
public class MyBean implements java.io.Serializable
{
 protected int theValue;
public MyBean()
     {
     }
  public void setMyValue(int newValue)
     {
        theValue = newValue;
      }
  public int getMyValue()
     {
      return theValue;
      }
}
```

4) Naming Pattern for Bean Properties and Events –

In this you will know basic rules for designing your own beans. First we want to know that there is no cosmic beans class that you extend to build your beans .Visual beans directly or indirectly extend the Component class. But non visual beans don't have to extend any particular superclass.

A bean is simply any class that can be manipulated in a builder tool.

The builder tool does not look at the superclass to determine the bean nature of a class but it analyzes the names of its methods. To enable this analysis, the method names for beans must follow certain patterns. The designers of the Java specification decided not to add keywords to the language to support visual programming.

Therefore they needed an alternative so that a builder tool could analyze a bean to learn its properties or events.
There are two alternatives mechanisms:

1) If the bean writer uses standard naming patterns for properties & events, then the builder tool can use the reflection mechanism to understand what properties & events the bean is supposed to expose.

2) The bean writer can supply a bean information class that tells the builder tool about the properties & events of the bean.

1) Naming pattern for properties is :

public Type getPropertyName()
public void setPropertyName(Type newValue)
corresponds to a read write property.

If u have a get method but not an associated set method, u define a read-only property.

If u have a set method without an associated get method then it defines a write only method.

There is one Exception to the get/set naming pattern. Properties that have boolean values should use an is/set naming pattern like:

public boolean isPropertyName()
public void setPropertyName(boolean b)

An animation might have a property running with two methods:

public boolean isRunning()
public void setRunning(boolean b)

The setRunning method would start & stop the animation.
The isRunning method would report its current status.
The bean analyzer performs a process called decapitalization to derive the property name.

2) Naming Pattern for Events is :

A bean builder envt. will infer that your bean generates events when you supply methods to add & remove eventlisteners.

All event class names must end with in Event, & the classes must extend the
EventObject class.

Suppose your bean generates events of type EventNameEvent. The listener interface must be called EventNameListener & the methods to add & remove a listener must be called.

public void addEventNameListener(EventNameListener e)
public void removeEventNameListener(EventNameListener e)

5) Bean Property Types –

A bean have a lots of diff. kinds of properties that it should expose in builder tool for a user to set at design time or get at run time.
It also triggers both standard & custom events.

The Java bean specification allows four types of Properties:

1) **Simple Properties –**

A simple property is a property that takes a single value such as a string or a number. Simple properties are easy to program. U use set/get naming convention in this property. we also have a read -only property of this bean with signature like :

public Dimension getPreferredSize() .

To implement a simple string property is he following:

```
public void setFileName(String f)
    {
        filename = f ;
        image= …….
        repaint();
    }
public String getFileName()
    {
        if (file==null)
        return null ;
        else return file.getPath() ;
    }
```

2) **Indexed Properties –**

An indexed property is one that gets or sets an array.
A chart bean uses an indexed property for data points.
With an indexed property u supply two pairs of get & set methods: one for the array & one for the individual entries.
They must follow this pattern:

a) *Type[ ] getPropertyName()*
b) *void setPropertyName(Type [ ] x)*

c) Type getPropertyName(int i)
d) void setPropertyName(int i ,Type x)

Here is an example of the indexed property :

```
public double[ ] getValues(int i)
 {
      if(0<= i && i<values.length)
      return values[i]
      return 0;
  }
public void setValues(int i,double value)
{
  if(0<= i && i<values.length)values [i] =value ;
}
```

3) Bound Properties –

Bound properties tell interested listeners that their value has changed. To implement a bound property, u must implement two mechanisms:

a) Whenever the value of the property changes, the bean must send a PropertyChangeEvent to all registered listeners. This change can occur when the set method is called or when the program user carries out an action such as editing text or selecting a file.

To enable interested listeners to register themselves, the bean has to implement the following two methods:

void addPropertyChangeListener(PropertyChangeListener listener)
void removePropertyChangeListener(PropertyChangeListener listener)

The java.beans package has a convenience class,called PropertyChangeSupport,that manages the listeners for you.

Private PropertyChangeSupport changeSupport = new PropertyChangeSupport(this) ;

Whenever the value of the property changes, use the firePropertyChange method of the PropertyChangeSupport object to deliver an event to all the registered listeners. That method has three parameters: the name of the property, the old value & the new value.

Ex:

changeSupport.firePropertyChange("running",false,true) ;

4) **Constraint properties –**

A constrained property is constrained by the fact that any listener can "veto" proposed changes, forcing it to revert to the old settings.One example is the closed property of the JInternalFrame class. If someone tries to call setClosed(true) on an internal frame, then all of its VetoableChangeListeners are notified. If any of them throws a PropertyVetoException, then the closed property is not changed & the set method throws a same exception. To build a constraint property, your bean must have the following methods to manage VetoableChangeListener objects:

public void addVetoable ChangeListener(VetoableChangeListener listener);
public void removeVetoable ChangeListener(VetoableChangeListener listener);

There is a convenience class,called VetaoableChangeSupport, that manages vetoable change listeners.Your bean should contain an object of this class:

private VetoableChangeSupport vetoSuport=new VetoableChangeSupport (this);

Adding & removing listeners should be delegated to this object:

```
public void addVetoable ChangeListener(VetoableChangeListener listener)
{
   vetoSupport.addVetoablechangeListener(listener) ;
}
```

```
public void removeVetoable ChangeListener(VetoableChangeListener listener)
{
   vetoSupport.removeVetoablechangeListener(listener) ;
}
```

6) BeanInfo Classes –

As your beans become complex, there may be features of your bean that naming patterns will not reveal. Many beans have get/set method pairs that should not correspond to bean properties.
Therefore the JavaBeans specification allows a far more flexible & powerful mechanism for storing information about your bean for use by a builder. You can define an object that implements the BeanInfo interface to describe your bean. When you implement this interface, a builder tool will look to the methods from the BeanInfo interface to tell it about the features that your bean supports. U need to follow a naming pattern to associate a BeanInfo object to the bean.The name of the bean Info class must be formed by adding BeanInfo to the name of the bean.

For Ex: the bean info class associated to the class ImageViewerBean must be named ImageViewerBeanBeanInfo.

The bean info class must be part of the same package as the bean itself. The most common reason for supplying a BeanInfo class is to gain control of the bean properties.
U construct a PropertyDescriptor for each property by supplying the name of the property & the class of the bean that contains it.

PropertyDescriptor descriptor = new PropertyDescriptor("filename",
ImageViewerBean.class);

Then implement the getPropertyDescriptors method of your BeanInfo class to return an array of all property descriptors.
The BeanInfo interface has four constants that cover the standard sizes :

*ICON_COLOR_16*32*
*ICON_COLOR_32*32*
*ICON_COLOR_16*16*
*ICON_COLOR_32*32*

7) Property Editors –

If u add an integer or string property to a bean, then that property is automatically displayed in the bean's property inspector. But what happens if u add a property whose values cannot easily be edited in a textfield, for eg, a date or a color ? Then u need to provide a separate component by which the user can specify the property value. Such components are called **property editors**.

For Ex: a property editor for a date object might be a calendar that lets the user scroll through the months & pick a date.
These property editors are registered with the property editor manager. The process of supplying a new property editor is :

1) *create a bean info class to accompany your bean.*
2) *Override the getPropertyDescriptors method that method returns an array of PropertyDescriptor objects.*
3) *U create one object for each property that should be displayed on a property editor.*
4) *U construct a PropertyDescriptor by supplying the name of the property & class of the bean that contains it :*

PropertyDescriptor descriptor= new PropertyDescriptor ("titlePosition", ChartBean.class);

5) *Then u call the setPropertyEditorClass method of the PropertyDescriptor class.*

descriptor.setPropertyEditorClass(TitlePositionEditor.class);

6) *Next u build an array of descriptors for properties of your bean.*

Ex: for the chart bean the descriptors for various properties are:

- A Color property, graphColor.
- A String property, title.
- An int property, titlePosition.
- A double [] property, values.

7) *The static registerEditor method of the PropertyEditorManager class sets a property editor for all properties of a given type. Ex:*

PropertyEditorManager.registerEditor(Date.class, CalendarSelector.class);

8) *Use the findEditor method in the PropertyEditorManager class to check whether a property editor exists for a given type in your builder tool. That method does the following:*

a) *It looks first to see which property editors are already registered with it.*

b) Then it looks for a class with a name that consists of the name of the type plus the word editor.

c) If neither looks up succeeds, then findEditor returns null.

For Ex: if a CalendarSelector class is registered for java.util.Date objects, then it would be used to edit a date property, otherwise a java.util.DateEditor would be searched.

Writing a Property Editor

Any property editor you write must implement the PropertyEditor interface, an interface with 12 methods.It is far more convenient to extend the convenience PropertyEditorSupport class that is supplied with the standard library. This support class comes with methods to add & remove property change listeners, and with default versions of all other methods of the PropertyEditor interface. For example,our editor for editing title position of a chart in our chart bean starts out like this:

```
II  property editor class for title position
  class TitlePositionEditor extends PropertyEditorSupport
  {
     ……
  }
```

Before writing a property editor, we should point out that the editor is under
the control of the builder, not the bean. The builder adheres to the following procedure to display the current value of the property:

1. It instantiates property editors for each property of the bean.
2. It asks the *bean* to tell it the current value of the property.
3. It then asks the *propertyeditor* to display the value.

The property editor can use either text-based or graphics-based methods to actually display the value.

Simple Property Editors -

Simple property editors work with text strings. You override the setAsText and getAsText methods. For example,our chart bean has a property that lets you set where the title should be displayed: Left, Center, or Right. These choices are implemented as integer constants.

```
private static final int LEFT= 0;
```

```
private static final int CENTER =1;
private static final int RIGHT= 2;
```

we define a property editor whose gelAsText method returns the value as a string. The method calls the, getValue method of the PropertyEditor to find the value of the property. Because this is a generic method, the value is returned as an Object. If the property type is a basic type, we need to return a wrapper object. In our case, the property type is int, and the call to getValue returns an Integer.

```
class TitlePositionEditor extends PropertyEditorSupport
 {
   public String getAsText()
    {
       int value =(Integer) getValue();
       return options[value];
    }
   private String[ ] options={ "Left" ,"Center" , "Right" };
}
```

Now, the text field displays one of these fields. When the user edits the textfield, this triggers a call to the setAsText method to update the property value by invoking the setValue method. It is a generic method whose parameter is of type Object.To set the value of a numeric type, we need to pass a wrapper object.

```
public void setAsText(String s)
{
   for (int i =0; i < options.length; i++)
    {
      if (options[i] .equals(s))
    {
       setValue(i);
       return;
    }
}
}
```

GUI - Based Property Editors –

More sophisticated property types cant be edited as text. Instead they are represented in two ways.

The property inspector contains a small area onto which the property editor will draw a graphical representation of the current value.

When the user clicks on that area, a custom editor dialog box pops up.

The dialog box contains a component to edit the property values, supplied by the property editor & various buttons, supplied by the builder envt.

To Build a GUI- Based property editor :

1) Tell the builder tool that u will paint the value & not use as a string.
2) "Paint" the value the user enters onto the GUI.
3) Tell the builder tool that u will be using a GUI based property editor.
4) Build the GUI.
5) Write the code to validate what the user tries to enter as the value.

For the first step, you override the getAsText method in the PropertyEditor interface to return null and the isPaintable method to return true.

```
public String getAsText()
 {
   return null;
 }
public boolean isPaintable()
  {
    return true;
  }
```

Then, you implement the paintValue method. It receives a Graphics context & the coordinates of the rectangle inside which you can paint.
Now, write the code that builds up the component that will hold the custom editor. For example, associated to our InverseEditor class is an InverseEditorPanel class that describes a GUI with two radio buttons to toggle between normal and inverse mode. However, the GUI actions must update the property values. We did this as follows:

1. *Have the custom editor constructor receive a reference to the property editor object and store it in a variable editor.*
2. *To read the property value, we have the custom editor call editor.getValue( ).*
3. *To set the object value, we have the custom editor. call editor.setValue(newValue) followed by editor.firePropertyChange().*

Customizers: -

- Customization provides a means for modifying the appearance & behavior of a bean within an application builder so it meets specific needs.

- Several levels of customization available for a bean developer to allow other developers to get maximized benefit from a bean's potential functionality.

A property editor, no matter how sophisticated, is responsible for allowing the user to set one property at a time. Especially if certain properties of a bean relate to each other, it may be more user friendly to give users a way to edit multiple properties at the same time. To enable this feature, you supply a customizer instead of (or in addition to) multiple property editors.
All customizers must:

1. *Extends java.awt.Component or one of its subclasses*
2. *Implements the java.bean.Customizer interface. This means implementing methods to register PropertyChangeListener objects, and firing Property change events at those listeners when a change to the target bean has occurred.*
3. *Implement a default constructor.*
4. *Associate the customizer with its target class via BeanInfo.getBeanDesccriptor.*