Beans

A Bean is a reusable software component with which we can create powerful applications and applets. Beans can provide support to the programmers to reuse and integrate the existing components. These components can be linked together to create applets, application or even new Beans for recess by others.

There are many builder tools that support to create beans. These provide a GUI based design environment to create and test the beans. The simple builder tool for creating Beans is **Beans Development Kit** (BDK) which can be downloaded freely from the Sun's java website. We can also develop the beans with **NetBeans** builder tool.

Once you install the BDK, you can create and test your beans. The BDK root directory contains the following sub-directories.

beanbox	Contains the run.bat file, which is used to load the GUI environment, and other folders.
jars	Contains the jar files for the default beans.
lib	Contains a jar file which is used to trace the methods.
doc	Contains documentation files.

The GUI environment provided by BDK contains 4 windows. They are: Tool Box, Bean Box, Properties Window and Method Tracer.

Tool Box:

When the BDK environment is opened, it automatically loads the toolbox with all the beans it finds within the jar files contained in jars directory. We can also load beans from jar files located else where by using load jar option of file menu of beanbox.

BeanBox

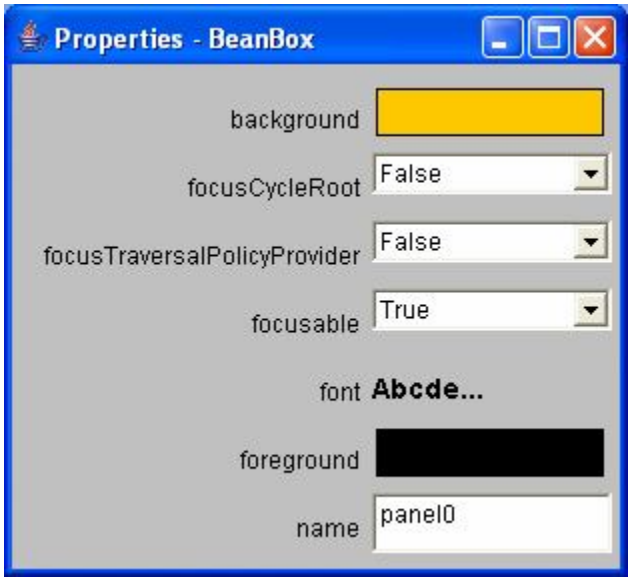
It is a container which provides the environment to test the beans. The menus of this Window are as follows.

File

Save	Saves the state of the beans currently in the BeanBox, including size, position,..
Make Applet	Creates an Applet for the BeanBox contents
Load	Loads the previously saved BeanBox
Load jar	Loads the bean jar files to the ToolBox
Print	Prints an image of the BeanBox contents
Clear	Clears the BeanBox container.
Exit	Closes the BeanBox.

Edit

Cut	Cuts the selected Bean.
Copy	Copies the selected Bean.
Paste	Pastes the previously cut or copied bean.
Events	Lists the selected bean's event firing methods.



Properties Window

This window shows the properties of the selected bean. We can customize the properties by making changes for the selected bean.

Method Tracer

This window shows the debugging messages and helps to trace the method calls.

The package **java.beans** conatins classes and interfaces that can support to create beans, edit the properties of them, bound a bean to other,...

Procedure to execute Java Beans

1. Compile the Bean Source Files.
2. Create a Manifest file as follows with name **BeanFile.mft**

   Name: BeanFile.class
   Java-Bean: True

3. Create a JAR file using the above manifest file and class file as follows

   jar -cvfm JAR-File.jar Manifest-File.mft Bean-File.class

-c	creates a new **J**ava **AR**chieve file
-v	displays the verbose output
-f	specifies the new archieve file name
-m	includes manifest information from a manifest file

4. Open **\BDK\beanbox** directory and run the batch file **run.bat.**
5. Select **loadjar** option from File menu of the **BeanBox window**.
6. Select the **JAR** file which you have created.
7. You will find that your Bean will be added to the **ToolBox**.
8. Click on the Bean you have created, from the ToolBox, and **drop it on BeanBox**. You will find, your bean will appear on the BeanBox.

A simple program on Beans to draw a rectangle.

```
import java.io.*;
import java.awt.*;

public class BeanTest extends Canvas implements Serializable
{
      public BeanTest()
      {
            setBackground(Color.green);
            setSize(50,50);
      }
}
```

→ Compile the above program and prepare a manifest file as follows.

Name: BeanTest.class
Java-Bean: True } **BeanTest.mft**

→ Create a jar file with the help of the above .class file and .mft file as follows.

jar -cvfm Test.jar BeanTest.mft BeanTest.class

→ Open BeanBox, chosse load jar option from the file menu and then select the **Test.jar** file which you have created.

→ The Bean will be loaded into the ToolBox. Click on the Bean. The mouse pointer changes to cross hair. Now drop the cursor on the BeanBox. Your bean will be added to the BeanBox.

Bean Properties:

The properties of a Bean can be categorized into 4 types. They are:

1. Simple Property :

Properties are the aspects of a Bean's appearance and behaviour that are changeable at design time. Properties are private values accessed through **getter** and **setter** methods.

2. Bound Property :

Some times, it may be necessary when a Bean property changes, another Bean object may want to be notified of the change and to react to the change. This is called a bound property. When ever a bound property changes, notification of the change is sent to interested listeners. A Bean containing a bound property must maintain a list of property change listeners and alert them when the bound property changes. To do this we should implement the **PropertyChangeListener** interface and extend **PropertyChangeSupport** class.

3. Constrained Property :

Constrained properties are so named because they provide a mechanism whereby a property can be **restricted to** a **certain range of values**. A Bean property is said to be constrained when any change to that property can be vetoed. A constrained property allows the listeners to veto (refuse or reject) changes by throwing PropertyVetoException. The major difference between bound and constrained properties is that **constarined properties** fire properety change events **before they are changed** and **bound properties** fire events **after they have been changed**.

4. Indexed Property :

Some times it is useful to have a **list of properties** in a Bean (like List component in AWT). Properties of a Bean of this sort are called indexed properties because they are accessed via an index into the list. Indexed properties are identified solely by the names of the **accessor** (a **setXXX()** method is called an accessor) and **mutator** (a **getXXX()** method is called a mutator) methods. The actual list may be stored in an array or any other indexeable data structure. These properties may be bound or constrained. The Bean class must implement the necessary add and remove listener methods and also fire property change events when an indexed property is being changed.

Program to add a simple property to the bean

```
import java.io.*;
import java.awt.*;
import java.beans.*;

public class BeanTest extends Canvas implements Serializable
{
        Color col=Color.green;
        public BeanTest()
        {
                setBackground(Color.green);
                setSize(100,100);
        }

        public void setColor(Color c)
        {
                col=c;
                repaint();
        }

        public Color getColor()
        {
                return col;
        }

        public void paint(Graphics g)
        {
                g.setColor(col);
                g.fillRect(10,10,20,20);
        }
}
```

Customization:

Customization provides a means for modifying the appearance and behavior of a bean within an application builder so it meets your specific needs. Customization is supported in two ways: by using **property editors** and bean **customizers.**

Property Editors:

It is a tool for customizing a particular property type. These are displayed in or activated from property sheets. A property sheet determines a peroperty's type, searches for a relevant property editor and displays the property's current value.

Property editors must implement the PropertyEditor interface, which provides methods that satisfy how a property should be displayed in a property sheet.

Customizer:

A Customizer is an application that specifically targets a Bean's customization. Sometimes properties are insufficient for representing a Bean's configurable attributes. Customizers are used where sophisticated instructions would be needed to change a Bean, and where property editors are too primitive to achieve bean customization.

All customizers must:

- extend java.awt.Component or one of its subclasses.
- implement the java.beans.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.
- implement a default constructor.
- associate the customizer with its target class via BeanInfo.getBeanDescriptor().

Introspection:

Introspection is the automatic process of analyzing a Bean's design patterns to reveal the Bean's properties, events, and methods. This process controls the publishing and discovery of Bean operations and properties.

The **BeanInfo** interface of the **java.beans** package defines a set of methods that allow bean implementors to provide explicit information about their beans. By specifying BeanInfo for a bean component, a developer can hide methods, specify an icon for the toolbox, provide descriptive names for properties, define which properties are bound properties, and much more.

The **Introspector** class provides descriptor classes with information about properties, events, and methods of a Bean. Methods of this class locate any descriptor information that has been explicitly supplied by the developer through BeanInfo classes. Then the Introspector class applies the naming conventions to determine what properties the Bean has, the events to which it can listen, and those which it can send.

Persistence:

A Bean has the property of persistence when its properties, fields, and state information are saved to and retrieved from storage. Component models provide a mechanism for persistence that enables the state of components to be stored in a non-volatile place for later retrieval.

Serialization:

The mechanism that makes persistence possible is called serialization. Object serialization means converting an object into a data stream and writing it to storage. Any applet, application, or tool that uses that Bean can then "**reconstitute"** it by **deserialization**. The object is then restored to its original state. When a Bean instance is serialized, it is converted into a data stream and is written to storage. All beans must persist. To persist, the beans must support serialization by implementing **Serializable** interface. The BeanBox writes serialized beans to a file with **.ser** extension. This can be done in 3 ways. They are:

1. Automatic Serialization
2. Selective Serialization
3. Complete Serialization

Automatic serialization:

The Serializable interface provides automatic serialization by using the Java Object Serialization tools. By marking our class with Serializable, we are telling Java Virtual Machine that we have make sure that our class will work with default serialization.

Selective Serialization:

When we do not want all fields to be serialized, we can use **transient** keyword to exclude fields from serilazation in a Serializable object. The default serialization will not serialize transient and static fields. We use the transient modifier to mark the fields as follows:

transient int quantity;

Complete Serilization:

Use the **Externalizable** interface when you need complete control over your Bean's serialization (for example, when writing and reading a specific file format). To use the Externalizable interface you need to implement two methods: **readExternal** and **writeExternal**. Classes that implement Externalizable must have a no-argument constructor.

Working with Events:

```
import java.awt.*;
import java.awt.event.*;
import java.beans.*;
import java.io.Serializable;

public class LabelBean extends Component implements Serializable
{
	String label="Press";

	public LabelBean()
	{
		setBackground(Color.lightGray);
		setSize(150,150);
	}

	public void setLabel(String lbl)
	{
		label=lbl;
		return;
	}
```

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

public void paint(Graphics g)
{
        int width = getSize().width;
        int height = getSize().height;
        g.setColor(getForeground());
        g.setFont(getFont());
        g.drawRect(2, 2, width - 4, height - 4);
        FontMetrics fm = g.getFontMetrics();
        g.drawString(label, (width - fm.stringWidth(label)) / 2,
                        (height + fm.getMaxAscent() - fm.getMaxDescent()) / 2);
}
}
```

Compile the above Bean and create a jar for that. Open BeanBox and load the jar of the above Bean. Click on Juggler Bean and drop it on BeanBox. Select LabelBean from ToolBox and create two instances of that Bean. Change the Label of one instance to Start and Stop for the other. Now select the Start LabelBean, choose Edit→events→mouse→mouseclicked from BeanBox Menubar. A Rubberband apperas. Drop this on Juggler Bean. Then you will be prompted to choose an event from **Event Target Dialog Box** of the Juggler Bean. Choose **start** and click Ok.

Now select Stop LabelBean and choose Edit→events→mouse→mouseclicked from BeanBox Menubar. A Rubberband appears. Drop this on Juggler Bean. Then you will be prompted to choose an event from Event Target Dialog Box of the Juggler Bean. Choose stop and click OK.

In both the cases, the BeanBox automatically prepares an **Adapter class** for the events. Now click on Stop LabelBean, the JugglerBean stops juggling. When you click on Start LabelBean it starts juggling.

Making Applets for the Beans:

You can make applets for the beans present in the BeanBox. To do this, select makeapplet Option from File Menu of BeanBox Menubar. Make proper settings in the **Make an Applet Dialog Box**. BeanBox automatically compiles and writes class files for the selected Bean. The default path for the generated files is **\beanbox\tmp\myApplet**. Just double click on the **.html** file in this directory. If your web browser is Java enabled, then the Bean will be displayed on the web page. Otherwise view this .html file with **appletviewer** utility which comes with JDK.

Saving the status of the Beanbox:

To save the status of the BeanBox, choose **save** option from File Menu of BeanBox window. You will be prompted to give a name for saving the status. Give a name for the file. Loading of this file automatically loads not only the Beans and also their status, that you have worked with previously when you save this file.