

JAVA BEANS

INTRODUCTION TO JAVA BEANS

Software components are self-contained software units developed according to the motto “Developed them once, run and reused them everywhere”. Or in other words, reusability is the main concern behind the component model.

A software component is a reusable object that can be plugged into any target software application. You can develop software components using various programming languages, such as C, C++, Java, and Visual Basic.

A “Bean” is a reusable software component model based on sun’s java bean specification that can be manipulated visually in a builder tool.

The term software component model describe how to create and use reusable software components to build an application

Builder tool is nothing but an application development tool which lets you both to create new beans or use existing beans to create an application.

To enrich the software systems by adopting component technology JAVA came up with the concept called Java Beans.

Java provides the facility of creating some user defined components by means of Bean programming.

We create simple components using java beans.

We can directly embed these beans into the software.

Advantages of Java Beans:

The java beans possess the property of “Write once and run anywhere”.

Beans can work in different local platforms.

Beans have the capability of capturing the events sent by other objects and vice versa enabling object communication.

The properties, events and methods of the bean can be controlled by the application developer.(ex. Add new properties)

Beans can be configured with the help of auxiliary software during design time.(no hassle at runtime)

The configuration setting can be made persistent.(reused)

Configuration setting of a bean can be saved in persistent storage and restored later.

What can we do/create by using JavaBean:

There is no restriction on the capability of a Bean.

It may perform a simple function, such as checking the spelling 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. One example of this is a button on a graphical user interface.

Software to generate a pie chart from a set of data points is an example of a Bean that can execute locally.

Bean that provides real-time price information from a stock or commodities exchange.

Definition of a builder tool:

Builder tools allow a developer to work with JavaBeans in a convenient way. By examining a JavaBean by a process known as Introspection, a builder tool exposes the discovered features of the JavaBean for visual manipulation. A builder tool maintains a list of all JavaBeans available. It allows you to compose the Bean into applets, application, servlets and composite components (e.g. a JFrame), customize its behavior and appearance by modifying its properties and connect other components to the event of the Bean or vice versa.

Some Examples of Application Builder tools:

--	--	--

TOOL	VENDOR	DESCRIPTION
		Java
Java Workshop2.0	Sun Micro Systems., Inc.,	Complete IDE that support applet, application and bean development Bean Oriented visual development toolset. Suit of bean oriented java development tool Supports only Beans development
Visual age for java	IBM	
Jbuilder	Borland Inc.	
Beans Development Kit	Sun Micro Systems., Inc.,	

JavaBeans Basic rules:

A JavaBean should:

- be public
- implement the Serializable interface
- have a no-arg constructor
- be derived from javax.swing.JComponent or java.awt.Component if it is visual

The classes and interfaces defined in the java.beans package enable you to create JavaBeans. The Java Bean components can exist in one of the following three phases of development

Construction phase

- Build phase

Execution phase

It supports the standard component architecture features of

Properties

Events

Methods

Persistence.

In addition Java Beans provides support for

Introspection (Allows Automatic Analysis of a java beans)

Customization (To make it easy to configure a java beans component)

Elements of a JavaBean:

Properties

Similar to instance variables. 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.

Methods

- Same as normal Java methods.
- Every property should have accessor (get) and mutator (set) method.
- All Public methods can be identified by the introspection mechanism.
- There is no specific naming standard for these methods

Events

Similar to Swing/AWT event handling.

The Java Bean Component Specification:

Customization:

Is the ability of JavaBean to allow its properties to be changed in build and execution phase.

Persistence:-

Is the ability of JavaBean to save its state to disk or storage device and restore the saved state when the JavaBean is reloaded

Communication:- Is the ability of JavaBean to notify change in its properties to other JavaBeans or the container.

Introspection:-

Is the ability of a JavaBean to allow an external application to query the properties, methods, and events supported by it. At the core of Java Beans is *introspection*. This is the process of analyzing a Bean to determine its capabilities. This is an essential feature of the Java Beans API because it allows another application, such as a design tool, to obtain information about a component. Without introspection, the Java Beans technology could not operate.

There are two ways in which the developer of a Bean can indicate which of its properties, events, and methods should be exposed. With the first method, simple naming conventions are used. These allow the introspection mechanisms to infer information about a Bean. In the second way, an additional class that extends the **BeanInfo** interface is provided that explicitly supplies this information. Both approaches are examined here.

Design Patterns for Properties

A *property* is a subset of a Bean's state. The values assigned to the properties determine the behavior and appearance of that component. A property is set through a *setter* method. A property is obtained by a *getter* method. There are two types of properties: simple and indexed.

Simple Properties

A simple property has a single value. 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 of these methods to access its values. A read-only property has only a get method. A write-only property has only a set method.

Here are three read/write simple properties along with their getter and setter methods:

```
private double depth, height, width;
```

```
public double getDepth( ) { return depth;
```

```
}
```

```
public void setDepth(double d) { depth = d;
```

```
}
```

```
public double getHeight( ) { return height;  
  
}
```

```
public void setHeight(double h) { height = h;  
  
}
```

```
public double getWidth( ) { return width;  
  
}
```

```
public void setWidth(double w) { width = w;  
  
}
```

Indexed Properties

An indexed property consists of multiple values. 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(int index);
```

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

```
public void setN(T values[ ]);
```

Here is an indexed property called **data** along with its getter and setter methods:

```
private double data[ ];
```

```
public double getData(int index) { return data[index];  
  
}
```

```

public void setData(int index, double value) { data[index] = value;

}

public double[ ] getData( ) { return data;

}

public void setData(double[ ] values) { data = new double[values.length];

System.arraycopy(values, 0, data, 0, values.length);

}

```

Design Patterns for Events

Beans use the delegation event model that was discussed earlier in this book. Beans 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 addTListener(TListener eventListener)
throws java.util.TooManyListenersException public void removeTListener(TListener eventListener)

```

These methods are used to add or remove a listener for the specified event. The version of **addTListener()** that does not throw an exception can be used to *multicast* an event, which means that more than one listener can register for the event notification. The version that throws **TooManyListenersException** *unicasts* the event, which means that the number of listeners can be restricted to one. In either case, **removeTListener()** is used to remove the listener. For example, assuming an event interface type called **TemperatureListener**, a Bean that monitors temperature might supply the following methods:

```

public void addTemperatureListener(TemperatureListener tl) {

...

}

public void removeTemperatureListener(TemperatureListener tl) {

```

...

}

Methods and Design Patterns

Design patterns are not used for naming nonproperty methods. The introspection mechanism finds all of the public methods of a Bean. Protected and private methods are not presented.

Using the BeanInfo Interface

As the preceding discussion shows, design patterns *implicitly* determine what information is available to the user of a Bean. The **BeanInfo** interface enables you to *explicitly* control what information is available. The **BeanInfo** interface defines several methods, including these:

```
PropertyDescriptor[] getPropertyDescriptors( )
EventSetDescriptor[] getEventSetDescriptors( )
MethodDescriptor[] getMethodDescriptors( )
```

They return arrays of objects that provide information about the properties, events, and methods of a Bean. The classes **PropertyDescriptor**, **EventSetDescriptor**, and **MethodDescriptor** are defined within the **java.beans** package, and they describe the indicated elements. By implementing these methods, a developer can designate exactly what is presented to a user, bypassing introspection based on design patterns.

When creating a class that implements **BeanInfo**, you must call that class *bname*BeanInfo, where *bname* is the name of the Bean. For example, if the Bean is called **MyBean**, then the information class must be called **MyBeanBeanInfo**.

To simplify the use of **BeanInfo**, JavaBeans supplies the **SimpleBeanInfo** class. It provides default implementations of the **BeanInfo** interface, including the three methods just shown. You can extend this class and override one or more of the methods to explicitly control what aspects of a Bean are exposed. If you don't override a method, then design-pattern introspection will be used. For example, if you don't override **getPropertyDescriptors()**, then design patterns are used to discover a Bean's properties

Services of JavaBean Components

Builder support:-Enables you to create and group multiple JavaBeans in an application.

Layout:-Allows multiple JavaBeans to be arranged in a development environment.

Interface publishing: Enables multiple JavaBeans in an application to communicate with each other.

Event handling:-Refers to firing and handling of events associated with a JavaBean.

Persistence:- Enables you to save the last state of JavaBean

Features of a JavaBean

- Support for “introspection” so that a builder tool can analyze how a bean works.
- Support for “customization” to allow the customisation of the appearance and behaviour of a bean.
- Support for “events” as a simple communication metaphor than can be used to connect up beans.
- Support for “properties”, both for customization and for programmatic use.
- Support for “persistence”, so that a bean can save and restore its customized state.

Beans Development Kit

Is a development environment to create, configure, and test JavaBeans.

The features of BDK environment are:

Provides a GUI to create, configure, and test JavaBeans.

Enables you to modify JavaBean properties and link multiple JavaBeans in an application using BDK.

Provides a set of sample JavaBeans.

Enables you to associate pre-defined events with sample JavaBeans.

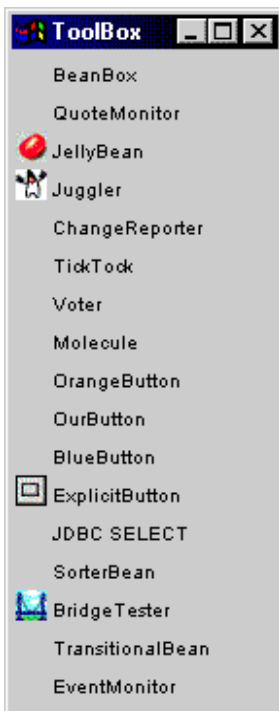
Identifying BDK Components

- Execute the run.bat file of BDK to start the BDK development environment.

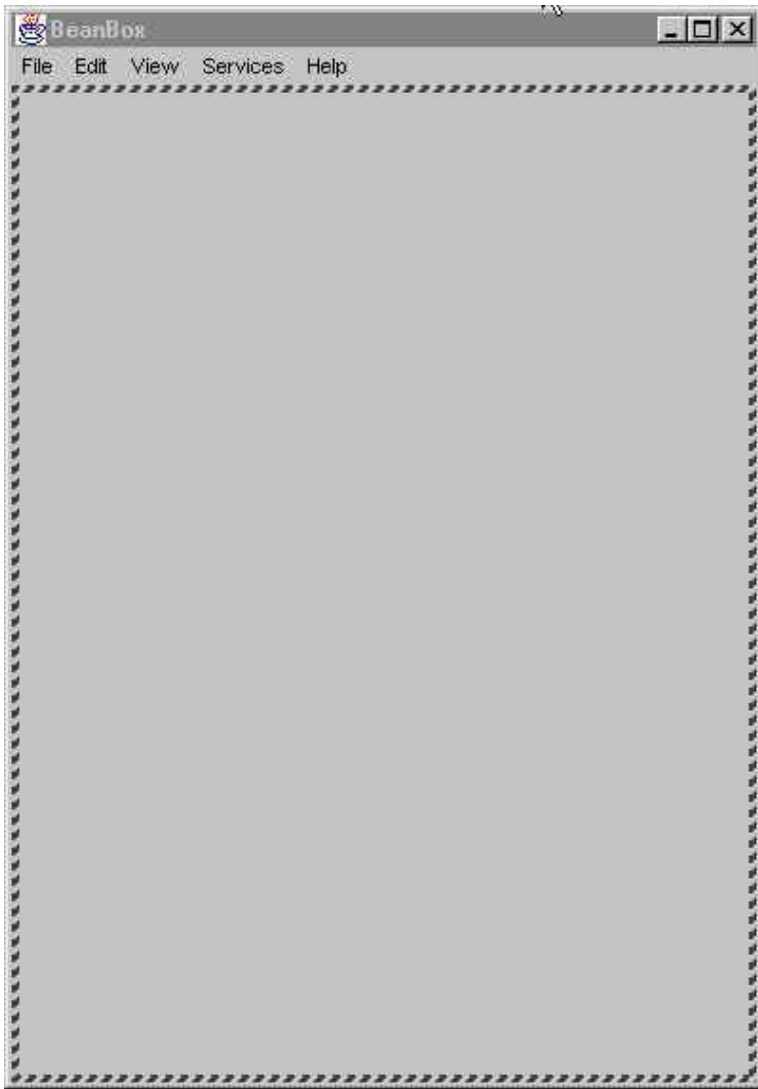
The components of BDK development environment are:

1. ToolBox
2. BeanBox
3. Properties
4. Method Tracer

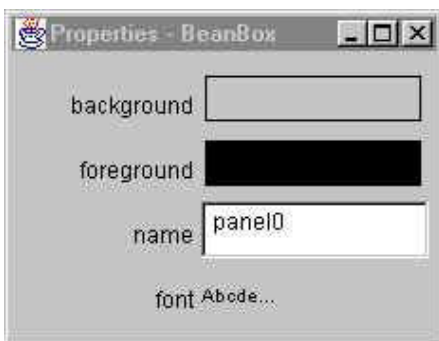
ToolBox window: Lists the sample JavaBeans of BDK. The following figure shows the ToolBox window



BeanBox window: Is a workspace for creating the layout of JavaBean application.



Properties window: Displays all the exposed properties of a JavaBean. You can modify JavaBean properties in the properties window. The following figure shows the Properties window



Method Tracer window: Displays the debugging messages and method calls for a JavaBean application. The following figure shows the Method Tracer window:



Steps to Develop a User-Defined JavaBean:

1. Create a directory for the new bean
2. Create the java bean source file(s)
3. Compile the source file(s)
4. Create a manifest file
5. Generate a JAR file
6. Start BDK
7. Load Jar file
8. Test.

1. Create a directory for the new bean

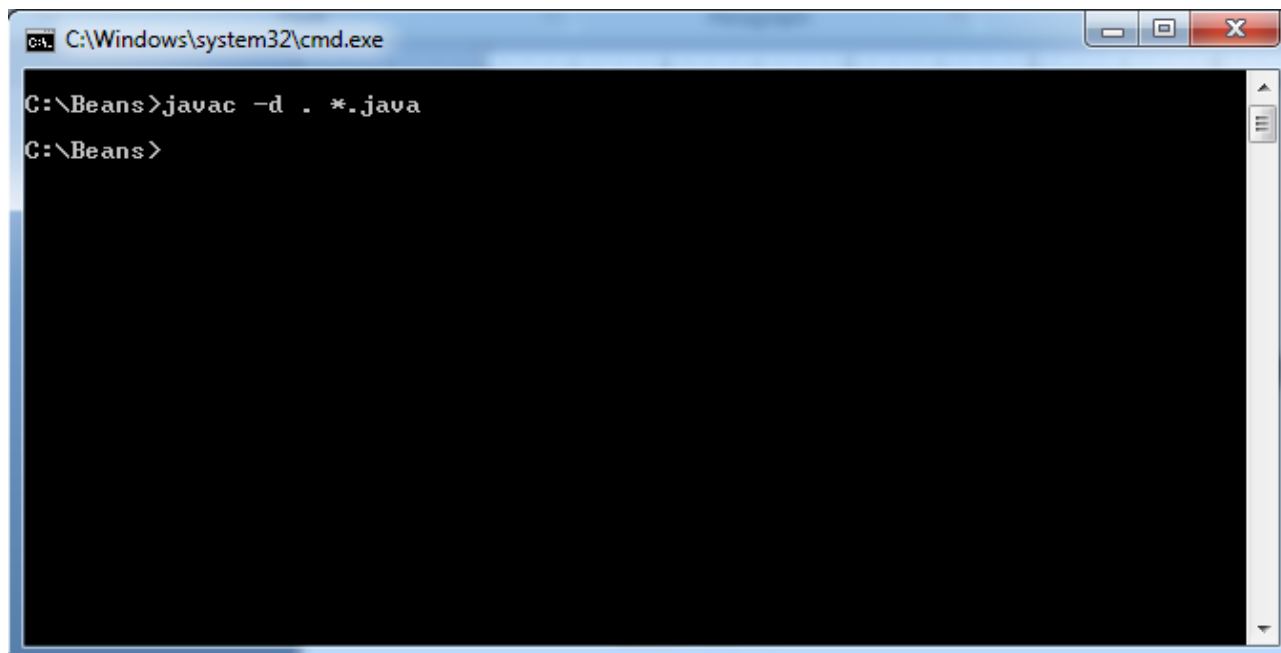
Create a directory/folder like C:\Beans

2. Create bean source file

```
package com.cmrcet.yellaswamy.beans;
import java.awt.*;
public class MyBean extends Canvas
{
    public MyBean()
    {
        setSize(70,50);
        setBackground(Color.green);
    }
}
```

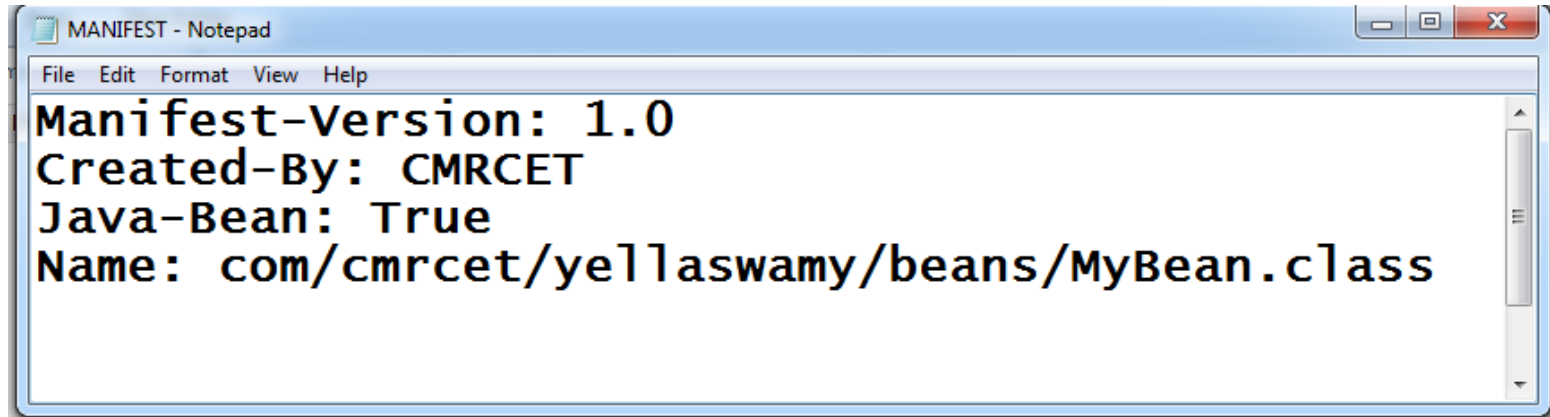
3. Compile the source file(s)

C:\Beans > javac -d . *.java



4. Create a manifest file Manifest File

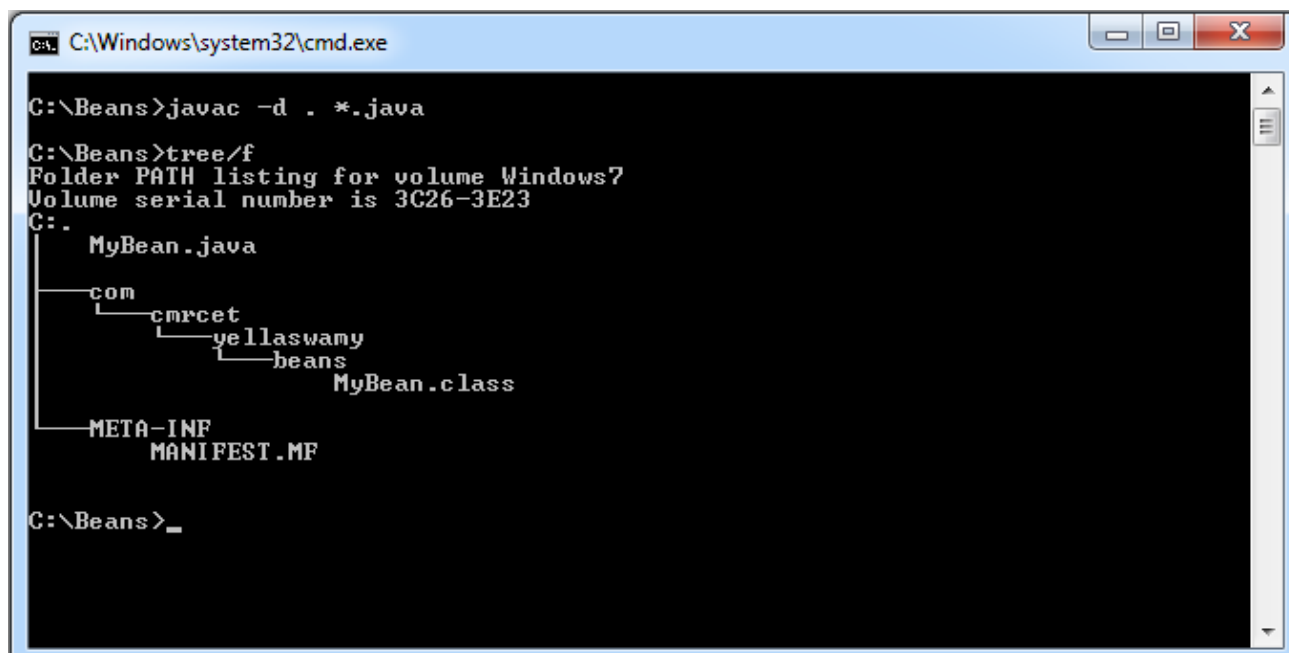
The manifest file for a JavaBean application contains a list of all the class files that make up a JavaBean. The entry in the manifest file enables the target application to recognize the JavaBean classes for an application. For example, the entry for the MyBean JavaBean in the manifest file is as shown:



Note: write that 2 lines code in the notepad and save that file as MANIFEST.MF in META-INF directory

The rules to create a manifest file are:

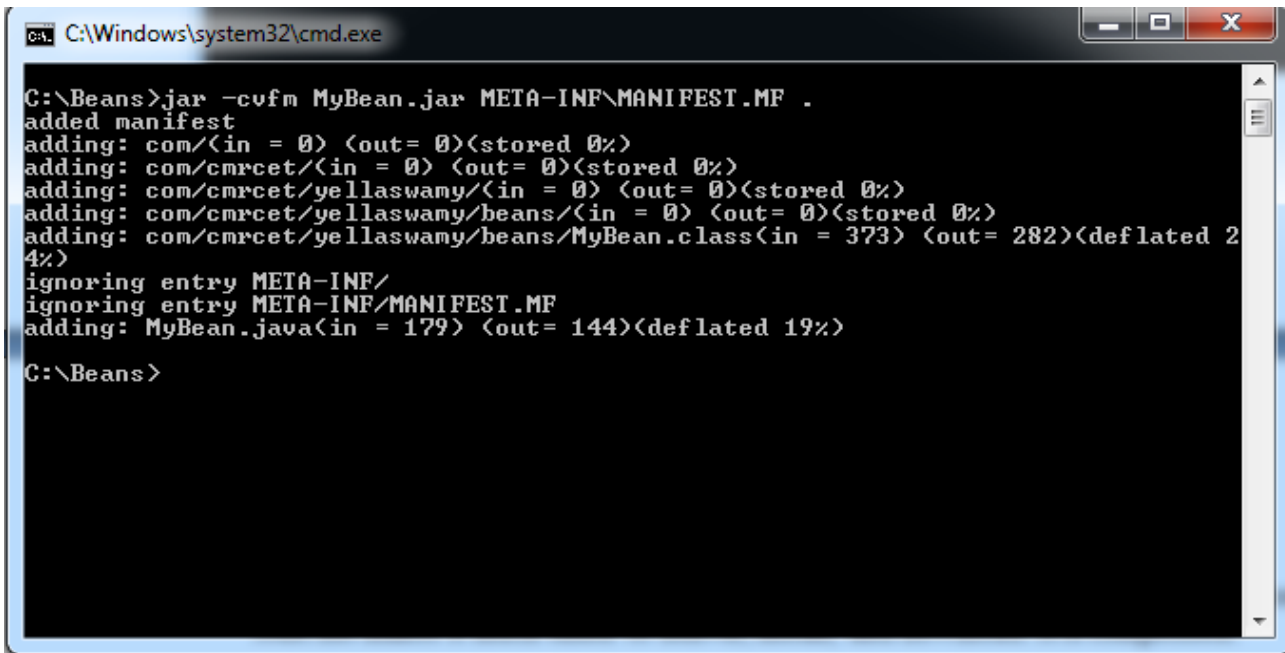
- Press the Enter key after typing each line in the manifest file.
- Leave a space after the colon.
- Type a hyphen between Java and Bean.
- No blank line between the Name and the Java-Bean entry.



5. Generate a JAR file

Syntax for creating jar file using manifest file

```
C:\Beans>jar -cvfm MyBean.jar META-INF\MANIFEST.MF
```



```
C:\Windows\system32\cmd.exe

C:\Beans>jar -cvfm MyBean.jar META-INF\MANIFEST.MF .
added manifest
adding: com/<in = 0> <out= 0><stored 0%>
adding: com/cmrcet/<in = 0> <out= 0><stored 0%>
adding: com/cmrcet/yellaswamy/<in = 0> <out= 0><stored 0%>
adding: com/cmrcet/yellaswamy/beans/<in = 0> <out= 0><stored 0%>
adding: com/cmrcet/yellaswamy/beans/MyBean.class<in = 373> <out= 282><deflated 24%>
ignoring entry META-INF/
ignoring entry META-INF/MANIFEST.MF
adding: MyBean.java<in = 179> <out= 144><deflated 19%>

C:\Beans>
```

JAR file:

JAR file allows you to efficiently deploy a set of classes and their associated resources.

JAR file makes it much easier to deliver, install, and download. It is compressed.

Java Archive File

- The files of a JavaBean application are compressed and grouped as JAR files to reduce the size and the download time of the files.
- The syntax to create a JAR file from the command prompt is:
- `jar <options><file_names>`
- The `file_names` is a list of files for a JavaBean application that are stored in the JAR file.

The various options that you can specify while creating a JAR file are:

c: Indicates the new JAR file is created.

f: Indicates that the first file in the `file_names` list is the name of the JAR file.

m: Indicates that the second file in the `file_names` list is the name of the manifest file.

t: Indicates that all the files and resources in the JAR file are to be displayed in a tabular format.

v: Indicates that the JAR file should generate a verbose output.

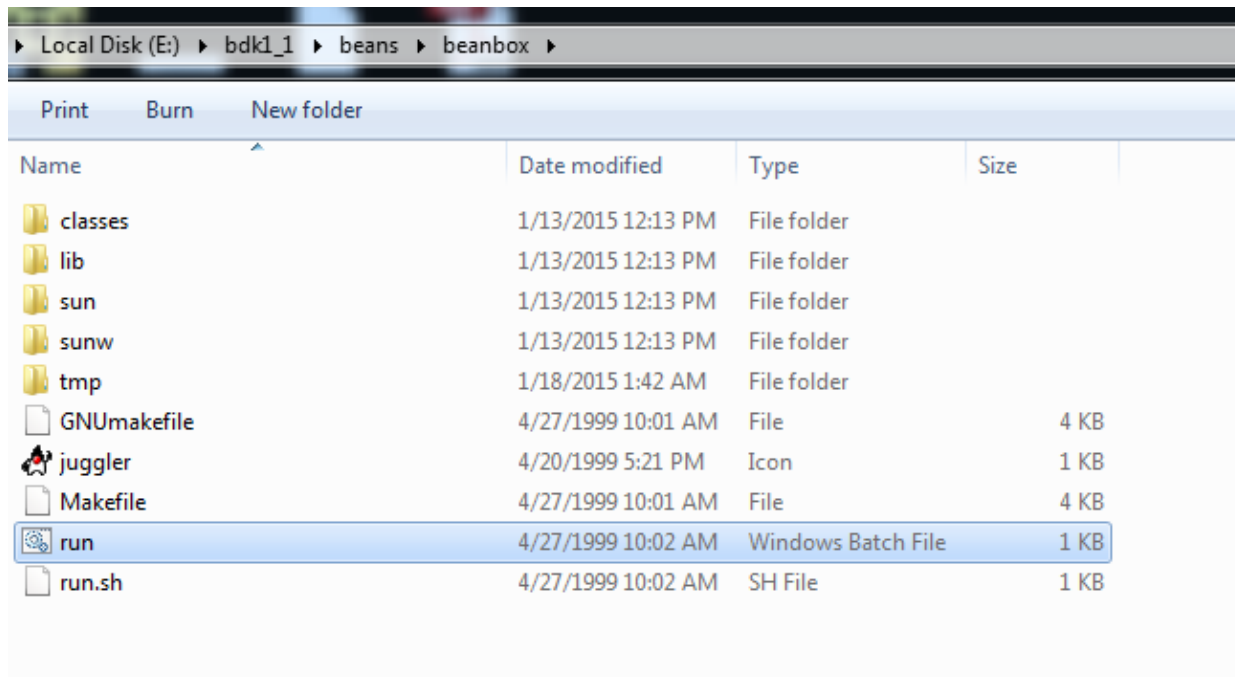
x: Indicates that the files and resources of a JAR file are to be extracted.

o: Indicates that the JAR file should not be compressed.

m: Indicates that the manifest file is not created.

Start BDK Go to

C:\bdk1_1\beans\beanbox Click on run.bat file.

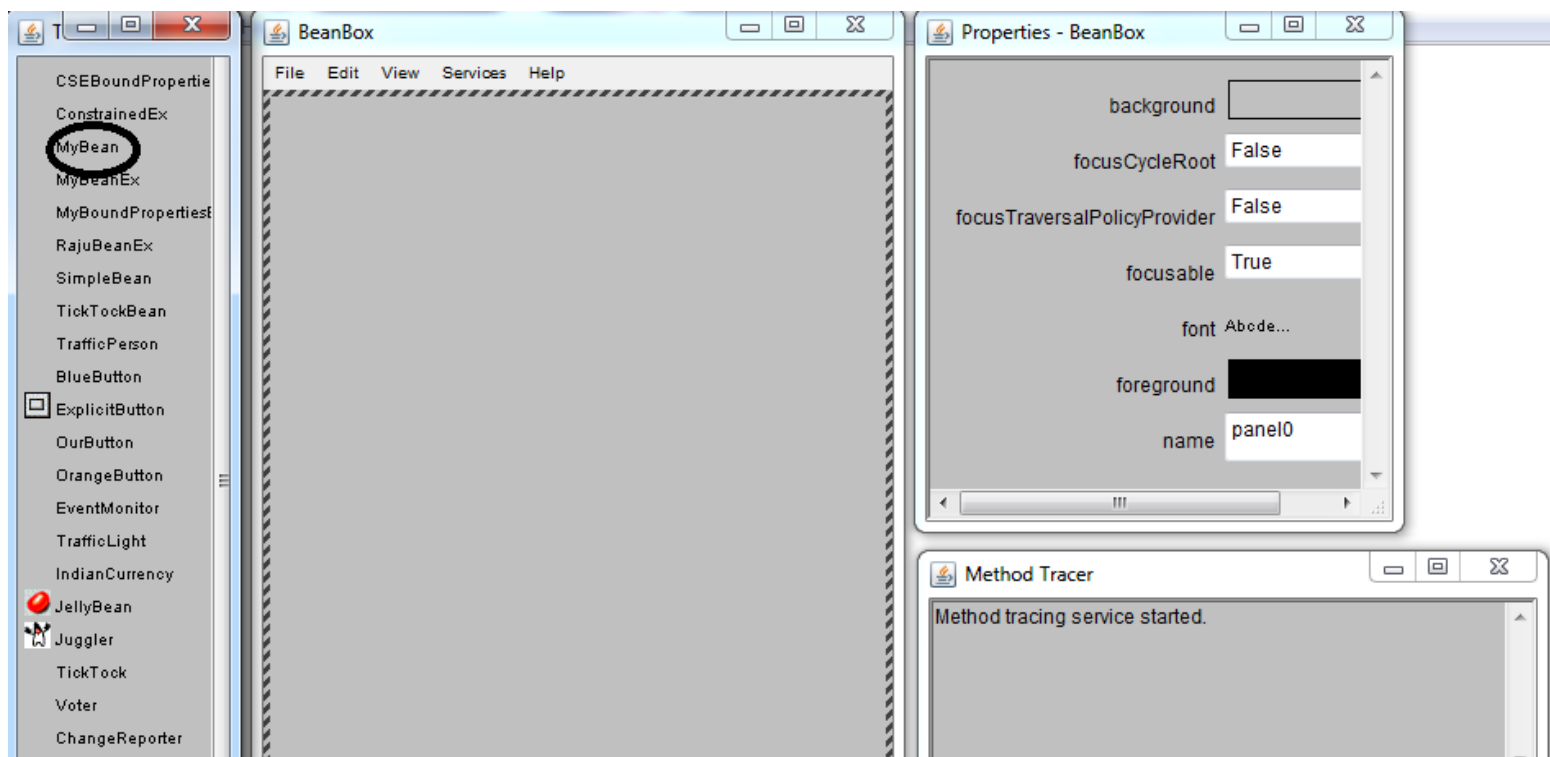


When we click on run.bat file the BDK software automatically started.

7. Load Jar file Go to

Beanbox->File->Load jar

. Here we have to select our created jar file when we click on ok, our bean(userdefined) MyBean appear in the ToolBox.



8. Test our created user defined bean

Select the MyBean from the ToolBox when we select that bean one + simple appear then drag that Bean in to the Beanbox. If you want to apply events for that bean, now we apply the events for that Bean.

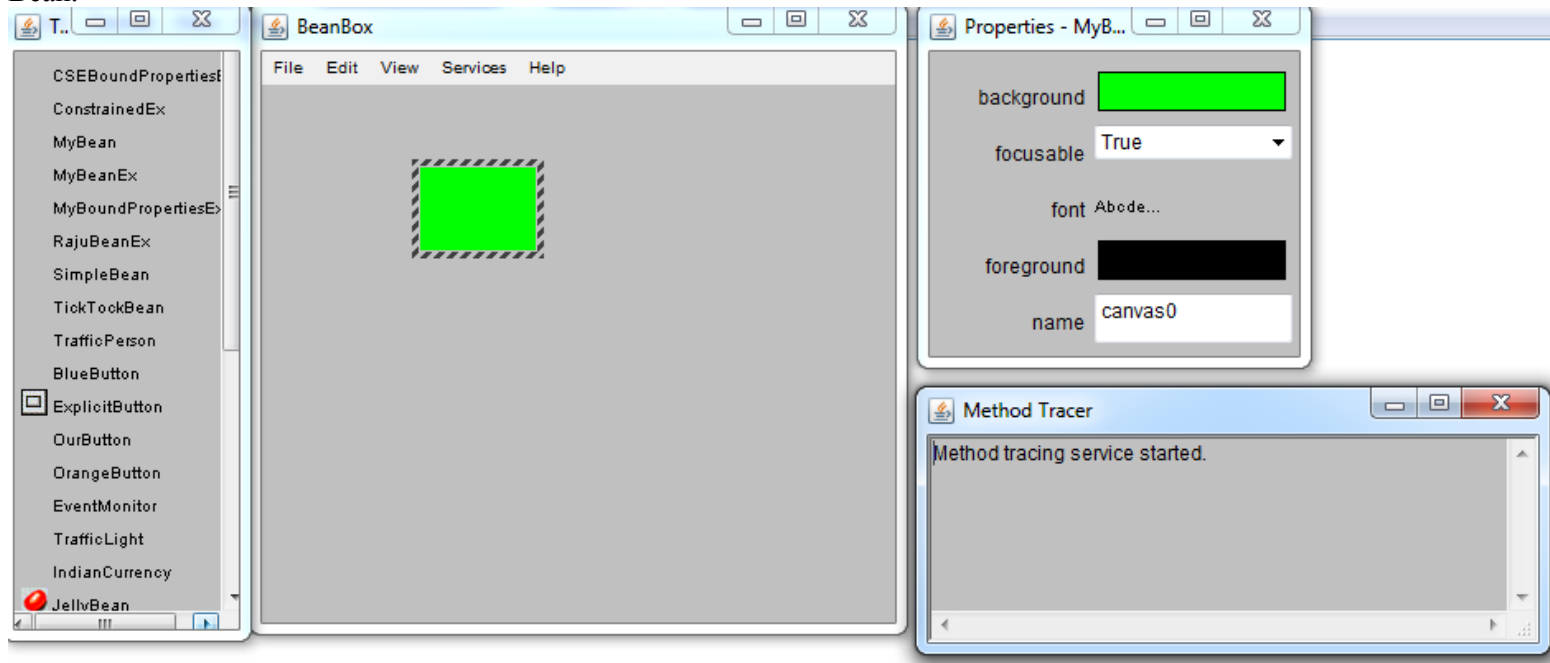
Introspection:

1. Introspection can be defined as the technique of obtaining information about bean properties, events and methods.
2. Basically introspection means analysis of bean capabilities.
3. Introspection is the automatic process by which a builder tool finds out which properties, methods, and events a bean supports.
4. Introspection describes how methods, properties, and events are discovered in the beans that you write.

5. This process controls the publishing and discovery of bean operations and properties Without introspection, the JavaBeans technology could not operate
6. Basically introspection means analysis of bean capabilities

There are two ways in which the developer of a Bean can indicate which of its properties, events, and methods should be exposed by an builder tool.

The first method, simple naming conventions are used. These allow the introspection mechanisms to infer information about a Bean.



Introspection:

1. Introspection can be defined as the technique of obtaining information about bean properties, events and methods.
2. Basically introspection means analysis of bean capabilities.
3. Introspection is the automatic process by which a builder tool finds out which properties, methods, and events a bean supports.
4. Introspection describes how methods, properties, and events are discovered in the beans that you write.
5. This process controls the publishing and discovery of bean operations and properties Without introspection, the JavaBeans technology could not operate
6. Basically introspection means analysis of bean capabilities

There are two ways in which the developer of a Bean can indicate which of its properties, events, and methods should be exposed by an builder tool.

The first method, simple naming conventions are used. These allow the introspection mechanisms to infer information about a Bean.

In the second way, an additional class is provided that explicitly supplies this information.

SimpleBean.java

```
//introspection
import java.beans.BeanInfo;
import java.beans.IntrospectionException;
import java.beans.Introspector;
import java.beans.*;
import java.beans.PropertyDescriptor;
public class SimpleBean
{
    private String name="CMRCET";
    private int Size;
    public String getName()
    {
```

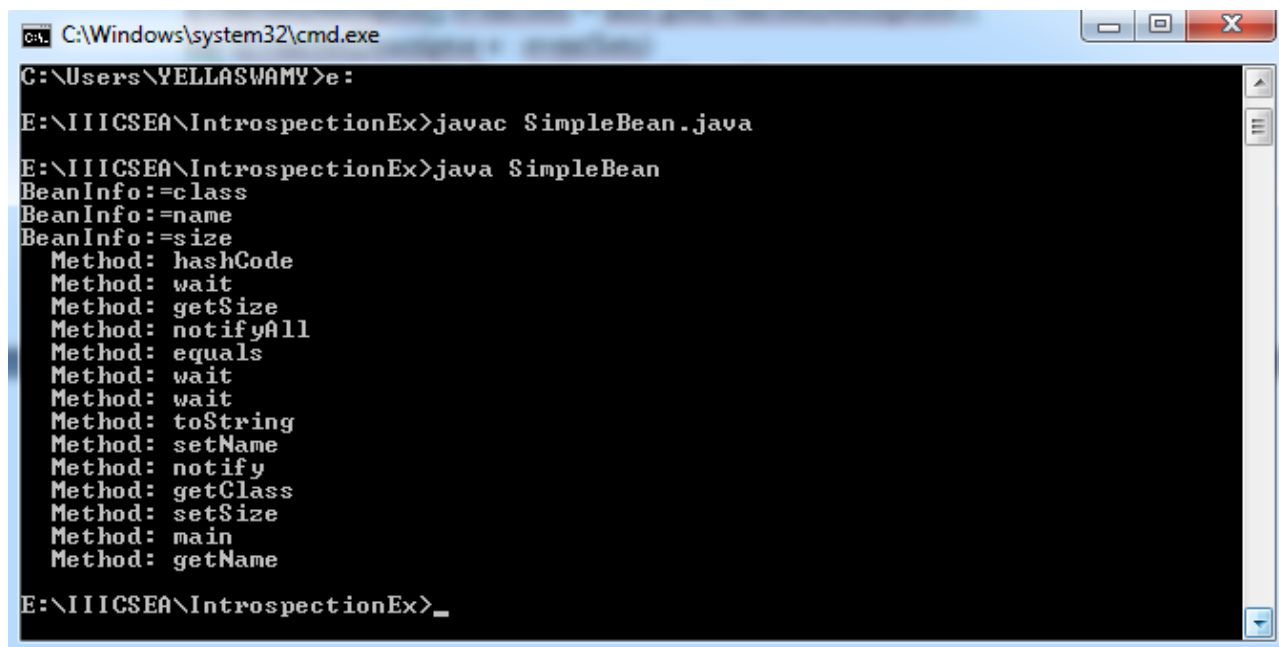
```

return this.name;
}
public int getSize()
{
return this.Size;
}
public void setSize(int size)
{
this.Size=size;
}
public void setName(String name)
{
this.name=name;
}
public static void main(String args[])throws IntrospectionException
{
BeanInfo info=Introspector.getBeanInfo(SimpleBean.class);
for(PropertyDescriptor pd:info.getPropertyDescriptors())
{
System.out.println("BeanInfo:="+pd.getName());
}
MethodDescriptor[] methods = info.getMethodDescriptors();
for (MethodDescriptor m : methods)
System.out.println(" Method: " + m.getName());
EventSetDescriptor[] eventSets = info.getEventSetDescriptors();
for (EventSetDescriptor e : eventSets)
System.out.println(" Event: " + e.getName());

}}

```

OUTPUT:



```

C:\Windows\system32\cmd.exe
C:\Users\YELLASWAMY>e:
E:\IIICSEA\IntrospectionEx>javac SimpleBean.java
E:\IIICSEA\IntrospectionEx>java SimpleBean
BeanInfo:=class
BeanInfo:=name
BeanInfo:=size
Method: hashCode
Method: wait
Method: getSize
Method: notifyAll
Method: equals
Method: wait
Method: wait
Method: toString
Method: setName
Method: notify
Method: getClass
Method: setSize
Method: main
Method: getName
E:\IIICSEA\IntrospectionEx>_

```

Design patterns for JavaBean Properties:-

A property is a subset of a Bean's state. 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. Properties are the private data members of the JavaBean classes. Properties are used to accept input from an end user in order to customize a JavaBean. Properties can retrieve and specify the values of various attributes, which determine the behavior of a JavaBean.

Types of JavaBeans Properties

- Simple properties
- Boolean properties
- Indexed properties
- Bound Properties
- Constrained Properties

Simple Properties:

Simple properties refer to the private variables of a JavaBean that can have only a single value. Simple properties are retrieved and specified using the get and set methods respectively.

1. A read/write property has both of these methods to access its values. The get method used to read the value of the property .The set method that sets the value of the property.

2. The setXXX() and getXXX() methods are the heart of the java beans properties mechanism. This is also called getters and setters. These accessor methods are used to set the property .

The syntax of get method is:

```
public return_type get<PropertyName>() public T getN(); public void setN(T arg)
```

N is the name of the property and T is its type

Ex:

```
public double getDepth()  
{  
    return depth;  
}
```

Read only property has only a get method.

The syntax of set method is:

```
public void set<PropertyName>(data_type value)
```

Ex:

```
public void setDepth(double d)  
{  
    Depth=d;  
}
```

Write only property has only a set method.

Boolean Properties:

A Boolean property is a property which is used to represent the values True or False. Have either of the two values, TRUE or FALSE. It can be identified by the following methods:

Syntax:

Let N be the name of the property and T be the type of the value then

```
public boolean isN();  
public void setN(boolean parameter);  
public Boolean getN();  
public boolean is<PropertyName>()  
public boolean get<PropertyName>()
```

First or second pattern can be used to retrieve the value of a Boolean.

```
public void set<PropertyName>(boolean value)
```

For getting the values isN() and getN() methods are used and for setting

Example:

```
public boolean dotted=false;  
public boolean isDotted() {  
    return dotted; }  
public void setDotted(boolean dotted) {  
    this.dotted=dotted; }
```

Indexed Properties:

Indexed Properties consist of multiple values. If a simple property can hold an array of values they are no longer called simple but instead indexed properties. The method's signature has to be adapted accordingly. An indexed property may expose set/get methods to read/write one element in the array (so-called 'index getter/setter') and/or so-called 'array getter/setter' which read/write the entire array.

Indexed Properties enable you to set or retrieve the values from an array of property values. Indexed Properties are retrieved using the following get methods: Syntax: public int[] get<PropertyName>()

Example:

```
private double data[];  
public double getData(int index)  
{  
    return data[index];  
}
```

Syntax: public property_datatype get<PropertyName>(int index)

Example:

```
public void setData(int index, double value)  
{  
    Data[index]=value;  
}
```


Indexed Properties are specified using the following set methods:

Syntax:

public void set<PropertyName>(int index, property_datatype value)

Example:

```
public double[] getData()  
{  
    return data;  
}
```

Syntax : public void set<PropertyName>(property_datatype[] property_array)

Example:

```
public void setData(double[] values)
{
}
```

The properties window of BDK does not handle indexed properties. Hence the output can not be displayed here.

Bound Properties:

A bean that has a bound property generates an event when the property is changed. Bound Properties are the properties of a JavaBean that inform its listeners about changes in its values. Bound Properties are implemented using the `PropertyChangeSupport` class and its methods. Bound Properties are always registered with an external event listener. The event is of type `PropertyChangeEvent` and is sent to objects that previously registered an interest in receiving such notifications bean with bound property - Event source Bean implementing listener -- event target.

In order to provide this notification service a JavaBean needs to have the following two methods:

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

`PropertyChangeListener` is an interface declared in the `java.beans` package. Observers which want to be notified of property changes have to implement this interface, which consists of only one method:

```
public interface PropertyChangeListener extends EventListener
{
    public void propertyChange(PropertyChangeEvent e );
}
```

Implementing Bound Properties:

The following steps are required to develop Bound Properties in a Java Bean

1. Add import statements to support Property Change Events
2. Instantiate a PropertyChangeSupport object
3. Add code to Fire, When a PropertyChangeEvent when the Property is changed.
4. Implement PropertyChangeSupport methods to add or remove listeners(the BeanBox will call these methods when a connection is made)
5. Build the JAR and Install it in the BeanBox
6. Test the Bean in BeanBox

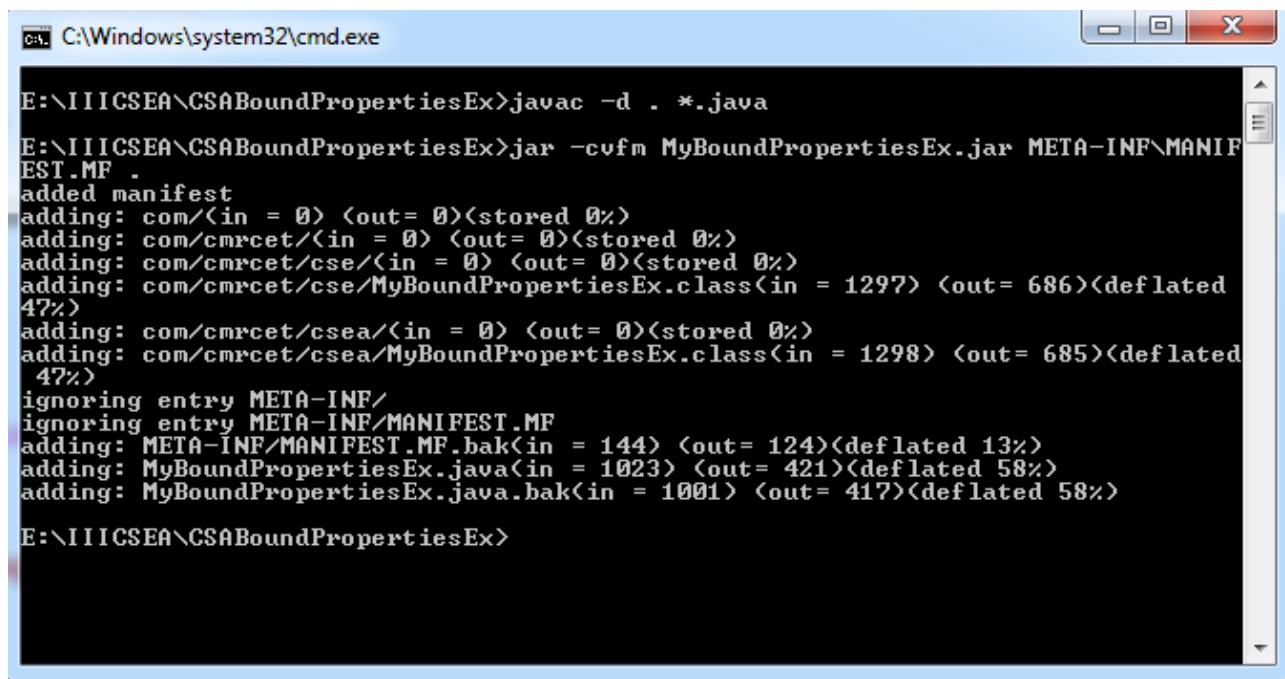
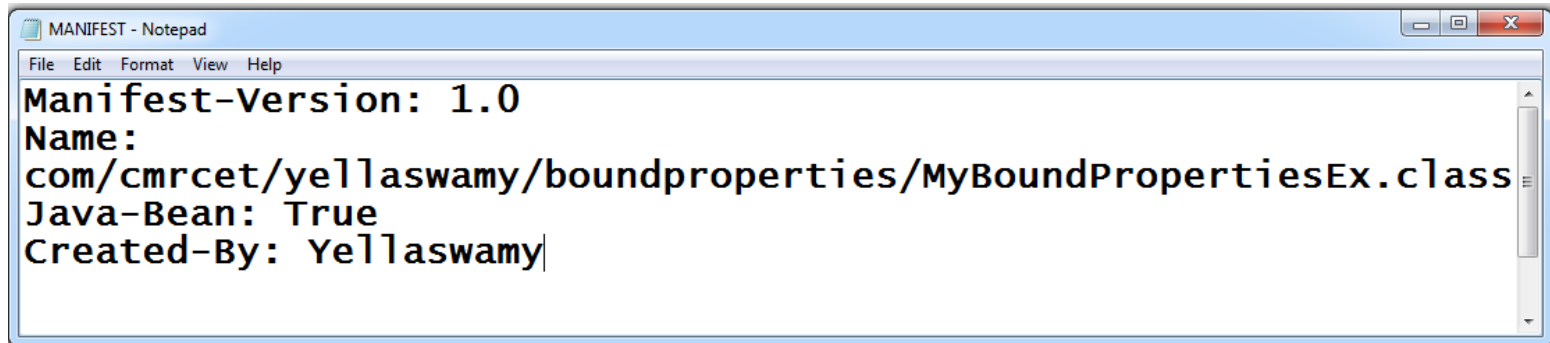
Java Program to create Bound Property in a Bean

```
// MyBoundPropertiesEx.java
package com.cmrcet.yellaswamy.boundproperties;
//step1
import java.beans.PropertyChangeSupport;
import java.io.Serializable;
import java.awt.*;
public class MyBoundPropertiesEx extends Canvas
{
//Step2:
String original="Welcome";
//instantiation
private PropertyChangeSupport pcs=new PropertyChangeSupport(this);
//constructor
public MyBoundPropertiesEx()
{
setBackground(Color.red);
setForeground(Color.blue);
}
public void setString(String newString)
{
//Step3
String oldString=original;
original=newString;
pcs.firePropertyChange("String","oldString","newString");
}
public String getString()
{
return original;
}
```

```

public Dimension getMinimumSize()
{
return new Dimension(50,50);
}
//Step4
public void addPropertyChangeListener(PropertyChangeListener l)
{
pcs.addPropertyChangeListener(l);
}
public void removePropertyChangeListener(PropertyChangeListener l)
{
pcs.removePropertyChangeListener(l);
}
}

```



```

E:\IIICSEA\CSABoundPropertiesEx>javac -d . *.java
E:\IIICSEA\CSABoundPropertiesEx>jar -cvfm MyBoundPropertiesEx.jar META-INF\MANIFEST.MF .
added manifest
adding: com/(in = 0) (out= 0)(stored 0%)
adding: com/cmrcet/(in = 0) (out= 0)(stored 0%)
adding: com/cmrcet/yellaswamy/(in = 0) (out= 0)(stored 0%)
adding: com/cmrcet/yellaswamy/boundproperties/(in = 0) (out= 0)(stored 0%)
adding: com/cmrcet/yellaswamy/boundproperties/MyBoundPropertiesEx.class(in = 1320) (out= 694)(deflated 47%)
ignoring entry META-INF/
ignoring entry META-INF/MANIFEST.MF

```

adding: MyBoundPropertiesEx.java(in = 1046) (out= 434)(deflated 58%)

E:\IICSEA\CSABoundPropertiesEx>tree/f

Folder PATH listing

Volume serial number is 3643-98F2

E:.

MyBoundPropertiesEx.jar
MyBoundPropertiesEx.java

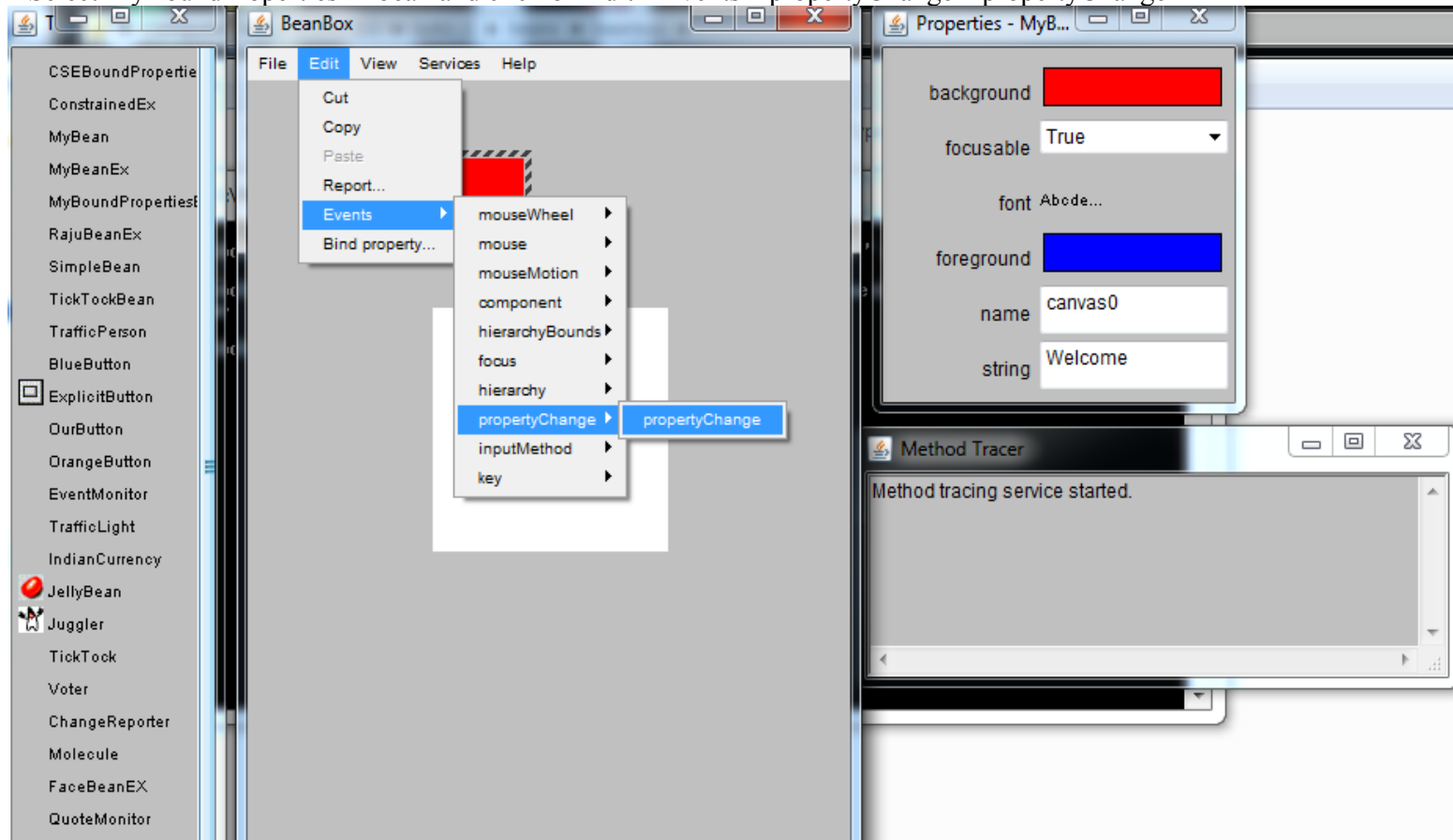
com
├── cmrcet
├── yellaswamy
└── boundproperties
MyBoundPropertiesEx.class

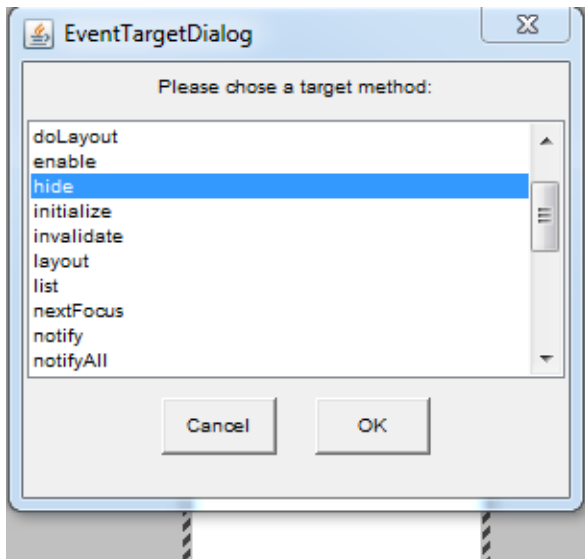
META-INF

MANIFEST.MF

Start BDK

- Checkout when MyBoundPropertiesEx Bean available in ToolBox or not.
- Instantiate a MyBoundPropertiesEx and any other Bean in the Beanbox.here I am selecting Molecule bean.
- Select MyBoundPropertiesEx bean and click on Edit→Events→propertyChange→propertyChange





After click on propertyChange method, connect MyBoundPropertiesEx bean with Molecule bean and select event to be generated on molecule bean when MyBoundPropertiesEx bean property is changed, here I am selecting “hide” event that is on changing MyBoundPropertiesEx bean property, molecule bean will be hide.

- On change MyBoundPropertiesEx bean property “Welcome” initial value to “Test” to any method.

Constrained Properties:

It generates an event when an attempt is made to change its value. Constrained Properties are implemented using the PropertyChangeEvent class. The event is sent to objects that previously registered an interest in receiving such notification. Those other objects have the ability to veto the proposed change. This allows a bean to operate differently according to the runtime environment.

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 Properties are the properties that are protected from being changed by other JavaBeans. Constrained Properties are registered with an external event listener that has the ability to either accept or reject the change in the value of a constrained property. Constrained Properties can be retrieved using the get method. The prototype of the get method is:

Syntax: `public String get<ConstrainedPropertyName>()` Can be specified using the set method.

The prototype of the set method is:

Syntax : `public String set<ConstrainedPropertyName>(String str) throws PropertyVetoException` There are three parts to constrained property Implementations:

- A source bean containing one or more constrained properties
- A PropertyChangeEvent object containing the property name, and its old and new values. This is the same class used for bound properties.
- Listener objects that implement the VetoableChangeListener interface. This object accepts or rejects proposed changes to a constrained property in the source Bean.

A bean Containing Constrained properties Must

- Allow VetoableChangeListener objects to register and unregister their interest in receiving notification that a property change is proposed.
- Fire property change events at those interested listeners when a property change is proposed. The event should be fired before the actual property change takes place. This gives each listener a chance to veto the proposed change. The PropertyChangeEvent is fired by a call to each listener's vetoableChange() method.
- If a listener vetoes, then make sure that any other listeners can revert to the old value.

Steps to implement Constrained Property

1. Add import Statement to support Property Change Events
2. Instantiate a VetoableChangeSupport object
3. Add code to Fire, When a PropertyChangeEvent when the property is changed.
4. Implement VetoableChangeSupport methods to add or remove listeners
5. Build the jar file and Install it in the Bean Box

6. Test this bean in BeanBox same as boundproperties

// **ConstrainedEx.java**

```
package com.yellaswamy.constrainedexample;
```

```
//step1
```

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

```
import java.beans.*;
```

```
public class ConstrainedEx extends Canvas
```

```
{
```

```
//Step2
```

```
String price="100";
```

```
VetoableChangeSupport vcs=new VetoableChangeSupport(this);
```

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

```
//constructor
```

```
public ConstrainedEx()
```

```
{
```

```
setBackground(Color.red);
```

```
setForeground(Color.blue);
```

```
}
```

```
public void setString(String newprice)
```

```
{
```

```
//Step3
```

```
String oldprice=price;
```

```
price=newprice;
```

```
pcs.firePropertyChange("price","oldprice","newprice");
```

```
}
```

```
public String getString()
```

```
{
```

```
return price;
```

```
}
```

```
public Dimension getMinimumSize()
```

```
{
```

```
return new Dimension(100,100);
```

```
}
```

```
//step4
```

```
public void addVetoableChangeListener(VetoableChangeListener vcl)
```

```
{
```

```
vcs.addVetoableChangeListener(vcl);
```

```
}
```

```
public void removeVetoableChangeListener(VetoableChangeListener vcl)
```

```
{
```

```
vcs.removeVetoableChangeListener(vcl);
```

```
}
```

```
}
```

Design Patterns for Events:

Handling Events in JavaBeans:

Enables Beans to communicate and connect together. Beans generate events and these events can be sent to other objects.

Event means any activity that interrupts the current ongoing activity.

Example: mouse clicks, pressing key...

User-defined JavaBeans interact with the help of user-defined events, which are also called custom events. You can use the Java event delegation model to handle these custom events.

The components of the event delegation model are:

Event Source: Generates the event and informs all the event listeners that are registered with it

Event Listener: Receives the notification, when an event source generates an event

Event Object: Represents the various types of events that can be generated by the event sources.

Creating Custom Events:

The classes and interfaces that you need to define to create the custom JavaBean events are:

- An event class to define a custom JavaBean event.
- An event listener interface for the custom JavaBean event.

- An event handler to process the custom JavaBean event.
- A target Java application that implements the custom event.

Creating the Event Class:

The event class that defines the custom event extends the EventObject class of the java.util package.

For example,

```
public class NumberEvent extends EventObject
{
    Public int number1,number2;
    Public NumberEvent(Object o,int number1,int number2)
    {
        super(o);
        this.number1=number1;
        this.number2=number2;
```



```
}}

```

Beans can generate events and send them to other objects.

Creating Event Listeners

When the event source triggers an event, it sends a notification to the event listener interface.

The event listener interface implements the `java.util.EventListener` interface.

Syntax:

```
public void addEventListener(TListenerEventListener);
public void addEventListener(TListenerEventListener) throws TooManyListeners;
public void removeEventListener(TListenerEventListener);

```

The target application that uses the custom event implements the custom listener.

For example,

```
public interface NumberEnteredListener extends EventListener
{
    public void arithmeticPerformed(NumberEvent mec);
}

```

Creating Event Handler

Custom event handlers should define the following methods:

- **addXXListener():** Registers listeners of a JavaBean event.
- **fireXX():** Notifies the listeners of the occurrence of a JavaBean event.
- **removeXXListener():** Removes a listener from the list of registered listeners of a JavaBean.

The code snippet to define an event handler for the custom event NumberEvent is:

```
public class NumberBean extends JPanel implements ActionListener
{
    public NumberBean()
    {}
    NumberEnteredListener mel;
    public void addNumberListener(NumberEnteredListener mel)
    {
        this.mel = mel;
    }
}

```

BeanDescriptor

A `BeanDescriptor` provides global information about a "bean", including its Java class, its `displayName`, etc

Beans

This class provides some general purpose beans control methods

DefaultPersistenceDelegate

The `DefaultPersistenceDelegate` is a concrete implementation of the abstract `PersistenceDelegate` class and is the delegate used by default for classes about which no information is available

Encoder

An `Encoder` is a class which can be used to create files or streams that encode the state of a collection of JavaBeans in terms of their public APIs.

EventHandler

The `EventHandler` class provides support for dynamically generating event listeners whose methods execute a simple statement involving an incoming event object and a target object.

EventSetDescriptor

An `EventSetDescriptor` describes a group of events that a given Java bean fires.

Expression

An `Expression` object represents a primitive expression in which a single method is applied to a target and a set of arguments to return a result - as in `"a.getFoo()"`.

FeatureDescriptor	The FeatureDescriptor class is the common baseclass for PropertyDescriptor, EventSetDescriptor, and MethodDescriptor, etc.
IndexedPropertyChangeEvent	An "IndexedPropertyChange" event gets delivered whenever a component that conforms to the JavaBeans specification (a "bean") changes a bound indexed property.
IndexedPropertyDescriptor	An IndexedPropertyDescriptor describes a property that acts like an array and has an indexed read and/or indexed write method to access specific elements of the array.
Introspector	The Introspector 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	The ParameterDescriptor class allows bean implementors to provide additional information on each of their parameters, beyond the low level type information provided by the java.lang.reflect.Method class.
PersistenceDelegate	The PersistenceDelegate class takes the responsibility for expressing the state of an instance of a given class in terms of the methods in the class's public API.
PropertyChangeEvent	A "PropertyChange" event gets delivered whenever a bean changes a "bound" or "constrained" property.
PropertyChangeListenerProxy	A class which extends the EventListenerProxy specifically for adding a named PropertyChangeListener.
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 to help build property editors.
PropertyEditorSupport	This is a support class to help build property editors.
SimpleBeanInfo	This is a support class to make it easier for people to provide BeanInfo classes.
Statement	A Statement object represents a primitive statement in which a single method is applied to a target and a set of arguments - as in "a.setFoo(b)".
VetoableChangeListenerProxy	A class which extends the EventListenerProxy specifically for associating a VetoableChangeListener with a "constrained" property.
VetoableChangeSupport	This is a utility class that can be used by beans that support constrained properties.
XMLDecoder	The XMLDecoder class is used to read XML documents created using the XMLEncoder and is used just like the ObjectInputStream.
XMLEncoder	The XMLEncoder class is a complementary alternative to the ObjectOutputStream and can be used to generate a textual representation of a <i>JavaBean</i> in the same way that the ObjectOutputStream can be used to create binary representation of Serializable objects.

The BeanInfoInterface :

By default an Introspector uses the Reflection API to determine the features of a *JavaBean*. However, a *JavaBean* can provide its own *BeanInfo* which will be used instead by the Introspector to determine the discussed information. This allows a developer hiding specific properties, events and methods from a builder tool or from any other tool which uses the Introspector class. Moreover it allows supplying further details about events/properties/methods as you are in charge of

creating the descriptor objects. Hence you can, for example, call the `setShortDescription()` method to set a descriptive description. A `BeanInfo` class has to be derived from the `SimpleBeanInfo` class and its name has to start with the name of the associated `JavaBean`. At this point it has to be underlined that the name of the `BeanInfo` class is the only relation between a `JavaBean` and its `BeanInfo` class.

The `BeanInfo` interface provides the methods that enable you to specify and retrieve the information about a `JavaBean`. The following table lists some of the methods of `BeanInfo` interface:

Method	Description
<code>MethodDescriptor[] getMethodDescriptors()</code>	Returns an array of the method descriptor objects of a <code>JavaBean</code> . The method descriptor objects are used to determine information about the various methods defined in a <code>JavaBean</code> .
<code>EventDescriptor[] getEventDescriptors()</code>	Returns an array of the event descriptor objects of a <code>JavaBean</code> . The event descriptor objects determine the information about the events associated with a <code>JavaBean</code> .
Method	Description
<code>PropertyDescriptor[] getPropertyDescriptors()</code>	Returns an array of the property descriptor objects of a <code>JavaBean</code> . The property descriptor objects are used to determine information about the various custom properties of a <code>JavaBean</code> .
<code>Image getIcon(int icon_Type)</code>	Returns a corresponding image object for one of the fields of the <code>BeanInfo</code> interface passed as a parameter to this method. The <code>BeanInfo</code> interface defines <code>int</code> fields, such as <code>ICON_COLOR_32x32</code> and <code>ICON_MONO_32x32</code> to represent icons.

The BeanBox Menus

This section explains each item in the BeanBox File, Edit, and View menus.

File Menu

Save : Saves the Beans in the BeanBox, including each Bean's size, position, and internal state. The saved file can be loaded via File|Load

SerializeComponent... Saves the Beans in the BeanBox to a serialized (.ser) file. This file must be put in a .jar file to be useable.

MakeApplet... Generates an applet from the BeanBox contents.

Load... Loads Saved files into the BeanBox. Will not load .ser files.

LoadJar... Loads a Jar file's contents into the ToolBox.

Print Prints the BeanBox contents.

Clear Removes the BeanBox contents.

Exit Quits the BeanBox without offering to save

Edit Menu

Cut Removes the Bean selected in the BeanBox. The cut Bean is serialized, and can then be pasted.

Copy Copies the Bean selected in the BeanBox. The copied Bean is serialized, and can then be pasted.

Paste Drops the last cut or copied Bean into the BeanBox.

Report... Generates an introspection report on the selected Bean.

Events Lists the event-firing methods, grouped by interface.

Bind property... Lists all the bound property methods of the selected Bean.

View Menu

Disable Design Mode

Removes the ToolBox and the Properties sheet from the screen. Eliminates all 14 beanBox design and test behavior (selected Bean, etc.), and makes the BeanBox behave like an application.

Hide Invisible Beans

Enterprise Java Beans

1. Remote Interface:

This should be a subtype of javax.ejb.EJBObject.

Declares all the business logic methods.

2. Home Interface:

This should be a subtype of javax.ejb.EJBHome.

Declares lifecycle methods, includes method such as create() that returns our remote interface type of object.

NOTE:

The javax.ejb.EJBObject and javax.ejb.EJBHome interfaces are subtypes of java.rmi.Remote

3. Bean class(or ejb class):

This class should not implement either the remote or even home interfaces.

This should implement one of the subtype of javax.ejb.EnterpriseBean interface, which is not related to java.rmi.Remote.

This class object is not a network enabled

This class object is not a network enabled object.

All the methods declared in our remote interface should be implemented in bean class

4. EJB object:

It is a term used to refer an instance of Remote interface implementation class.

This is implemented by the container provider.

5. Home Object:

It is a term used to refer an instance of Home interface implementation class.

This implementation is also provided by container provider

Types of EJB:

1. Session Beans:

a) StatelessSessionBean(SLSB)

b) StatefulSessionBean(SFSB)

2. Entity Beans:

a) BeanManagedPersistence(BMP)

b) ContainerManagedPersistence(CMP)

3.MessageDrivenBean(MDB) (this is new in EJB2.0)

EJB ARCHITECTURE

