Unit-3

Java Beans

1 What Is a Java Bean?

- ✓ A **Java Bean** is a software component that has been designed to be **reusable** in a variety of different environments.
- ✓ 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.
- ✓ Beans are important, because they allow us to build complex systems from software components.
- ✓ These components may be provided by you or supplied by one or more different vendors.
- ✓ Java Beans defines an architecture that specifies how these building blocks can operate together.
- ✓ To better understand the value of Beans, consider the following. Hardware designers have a wide variety of components that can be integrated together to construct a system.
- ✓ Large applications grow in complexity and become very difficult to maintain and enhance.

JavaBeans is a portable, platform-independent model written in Java Programming Language. Its components are referred to as beans.

In simple terms, JavaBeans are classes which encapsulate several objects into a single object. It helps in accessing these object from multiple places.

JavaBeans contains several elements like Constructors, Getter/Setter Methods and much more.

JavaBeans has several conventions that should be followed:

- Beans should have a default constructor (no arguments)
- Beans should provide getter and setter methods
  - A *getter method* is used to read the value of a readable property
  - To update the value, a *setter method* should be called
- Beans should implement *java.io.serializable*, as it allows to save, store and restore the state of a JavaBean you are working on.

Components of JavaBeans

The classes that contained definition of beans is known as components of JavaBeans. These classes follows certain design conventions. It includes properties, events, methods and persistence. There are two types of components, GUI based and non GUI based. For instance JButton is example of a component not a class.

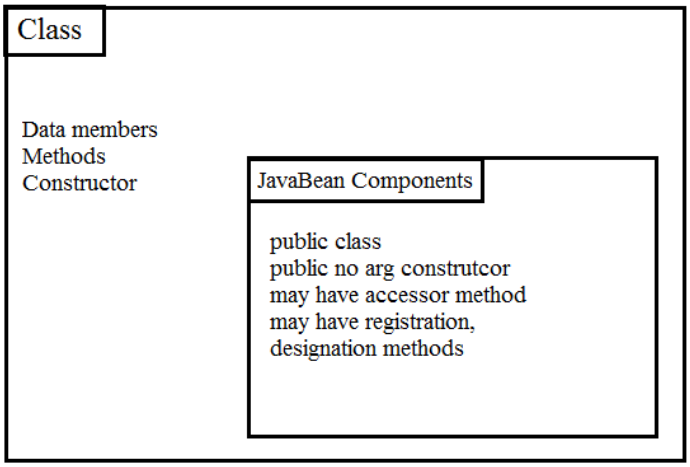
Properties (date members): Property is a named attribute of a bean, it includes color, label, font, font size, display size. It determines appearance, behavior and state of a bean.

Methods: Methods in JavaBeans are same as normal Java methods in a class. It doesn't follow any specific naming conventions. All properties should have accessor and getter methods.

Events: Events in JavaBeans are same as SWING/AWT event handling.

Persistence: Serializable interface enables JavaBean to store its state.

JavaBean has no argument constructor.



JavaBean component

2. Advantages of JavaBean

The following are the advantages of JavaBean:

- Bean obtains all the benefits of Java's "write-once, run-anywhere" paradigm.
- The JavaBean properties, event and methods can be exposed to another application.
- It provides an easiness to reuse the software components.
- Reusability in different environments.
- JavaBeans are dynamic, can be customized.
- Can be deployed in network systems
- A Bean may register to receive events from other objects and can generate events that are sent to other objects.
- The configuration settings of a Bean can be saved in persistent storage and restored at a later time.

Disadvantages of JavaBean

The following are the disadvantages of JavaBean:

- JavaBeans are mutable. So, it can't take advantages of immutable objects.
- Creating the setter and getter method for each property separately may lead to the boilerplate code.

3. Introspection

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

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.

3.a. Design Patterns for Properties

A JavaBean property is a named feature that can be accessed by the user of the object. The feature can be of any Java data type, containing the classes that you define.

A JavaBean property may be read, write, read-only, or write-only. JavaBean features are accessed through two methods in the JavaBean's implementation class:

1. getPropertyName ()

For example, if the property name is firstName, the method name would be getFirstName() to read that property. This method is called the accessor.

2. setPropertyName ()

For example, if the property name is firstName, the method name would be setFirstName() to write that property. This method is called the mutator.

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
T is its type

public T getN( )
public void setN(T *arg*)

Example:

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

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

public void setDepth(double depth) {
    this.depth = depth;
}

public double getHeight() {
    return height;
}

public void setHeight(double height) {
    this.height = height;
}

public double getWidth() {
    return width;
}

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

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
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);
}
```

3.b. Design Patterns for Events

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*)

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) {
...
}
```

3.c. 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.

3.d. Using the BeanInfo Interface

The **BeanInfo** interface enables to *explicitly* control what information is available.

The **BeanInfo** interface defines several methods, including these:

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

The classes **PropertyDescriptor, EventSetDescriptor**, and **MethodDescriptor** are defined within the **java.beans** package. By implementing these methods, a developer can presented to a user, bypassing introspection based on design patterns.

When creating a class that implements **BeanInfo**, We must call that class *bname*BeanInfo, where *bname* is the name of the Bean.

4. Bound and Constrained Properties

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 );
}
```

Constrained Properties:

- It generates an event when an attempt is made to change it value.
- Constrained Properties are implemented using the ***PropertyChangeEvent*** class.
- The event is sent to objects that previously registered an interest in receiving an such notification
- Those other objects have the ability to veto(reject) 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

5. Persistence

Persistence means an ability to save properties and events of our beans to non-volatile storage and retrieve later.

It has the ability *to save a bean to storage and retrieve it at a later time* Configuration settings are saved It is implemented by *Java serialization*.

If a bean inherits directly or indirectly from Component class it is automatically *Serializable*.

Transient keyword can be used to designate data members of a Bean that should not be serialized.

- Enables developers to customize Beans in an application builder, and then retrieve those Beans, with customized features intact, for future use,perhaps in another environment.
- Java Beans supports two forms of persistence:
    1. Automatic persistence
    2. External persistence

Automatic Persistence:
Automatic persistence are java's built-in serialization mechanism to save and restore the state of a bean.

External Persistence:
External persistence, on the other hand, gives you the option of supplying your own custom classes to control precisely how a bean state is stored and retrieved.

- ✓ Juggle Bean. (The art of moving objects)
- ✓ Building an applet
- ✓ Your own bean

6. Customizers:

A Bean developer can provide a *customizer* that helps another developer configure the Bean.

A customizer can provide a step-by-step guide through the process that must be followed to use the component in a specific context.

Online documentation can also be provided.

A Bean developer has great flexibility to develop a customizer that can differentiate his or her product in the marketplace.

7. The Java Beans API

The Java Beans functionality is provided by a set of classes and interfaces in the **java.beans** package.

Table 28- 1 lists the interfaces in **java.beans** and provides a brief description of their functionality.

Table 28-2 lists the classes in **java.beans**.

Set of classes and interfaces in the java.beans package **Package java.beans**

Contains classes related to developing *beans* -- components based on the JavaBeans™ architecture.

Interface	Description
AppletInitializer	Methods in this inter face are used to initialize Beans that are also applets.
BeanInfo	This inter face allows a designer to specify information about the proper ties, events, and methods of a Bean.
Customizer	This inter face allows a designer to provide a graphical user inter face through which a Bean may be configured.
DesignMode	Methods in this inter face determine if a Bean is executing in design mode.
ExceptionListener	A method in this inter face is invoked when an exception has occurred.
Proper tyChangeListener	A method in this inter face is invoked when a bound proper ty is changed.
Proper tyEditor	Objects that implement this inter face allow designers to change and display proper ty values.
VetoableChangeListener	A method in this inter face is invoked when a constrained proper ty is changed.
Visibility	Methods in this inter face allow a Bean to execute in environments where a graphical user inter face is not available.

TABLE 28-1 The Inter faces in java.beans

Class	Description
BeanDescriptor	This class provides information about a Bean. It also allows you to associate a customizer with a Bean.
Beans	This class is used to obtain information about a Bean.
DefaultPersistenceDelegate	A concrete subclass of PersistenceDelegate.
Encoder	Encodes the state of a set of Beans. Can be used to write this information to a stream.
EventHandler	Suppor ts dynamic event listener creation.
EventSetDescriptor	Instances of this class describe an event that can be generated by a Bean.
Expression	Encapsulates a call to a method that returns a result.
FeatureDescriptor	This is the superclass of the Proper tyDescriptor, EventSetDescriptor, and MethodDescriptor classes.
IndexedProper tyChangeEvent A	subclass of Proper tyChangeEvent that represents a change to an indexed proper ty.
IndexedProper tyDescriptor	Instances of this class describe an indexed proper ty of a Bean.
IntrospectionException	An exception of this type is generated if a problem occurs when analyzing a Bean.
Introspector	This class analyzes a Bean and constructs a BeanInfo object that describes the component.

MethodDescriptor	Instances of this class describe a method of a Bean.
ParameterDescriptor	Instances of this class describe a method parameter.
PersistenceDelegate	Handles the state information of an object.
Proper tyChangeEvent	This event is generated when bound or constrained proper ties are changed. It is sent to objects that registered an interest in these events and that implement either the Proper tyChangeListener or VetoableChangeListener inter faces.
Proper tyChangeListenerProxy	Extends EventListenerProxy and implements Proper tyChangeListener.
Proper tyChangeSuppor t	Beans that suppor t bound proper ties can use this class to notify Proper tyChangeListener objects.
Proper tyDescriptor	Instances of this class describe a proper ty of a Bean.
Proper tyEditorManager	This class locates a Proper tyEditor object for a given type.
Proper tyEditorSuppor t	This class provides functionality that can be used when writing proper ty editors.
Proper tyVetoException	An exception of this type is generated if a change to a constrained proper ty is vetoed.
SimpleBeanInfo	This class provides functionality that can be used when writing BeanInfo classes.
Statement	Encapsulates a call to a method.
VetoableChangeListenerProxy	Extends EventListenerProxy and implements VetoableChangeListener.
VetoableChangeSuppor t	Beans that suppor t constrained proper ties can use this class to notify VetoableChangeListener objects.
XMLDecoder	Used to read a Bean from an XML document.
XMLEncoder	Used to write a Bean to an XML document.

TABLE 28-2 The Classes in java.beans (continued)

Although it is beyond the scope of this chapter to discuss all of the classes, four are of particular interest: **Introspector, PropertyDescriptor, EventSetDescriptor**, and **MethodDescriptor**. Each is briefly examined here.

Introspector

The **Introspector** class provides several static methods that support introspection. Of most interest is **getBeanInfo( )**.

This method returns a **BeanInfo** object that can be used to obtain information about the Bean.

The **getBeanInfo( )** method has several forms, including the one shown here:

static BeanInfo getBeanInfo(Class<?> *bean*) throws IntrospectionException

The returned object contains information about the Bean specified by *bean.*

PropertyDescriptor

The **PropertyDescriptor** class describes the characteristics of a Bean property. It supports several methods that manage and describe properties.

For example, We can determine if a property is bound by calling **isBound( )**.
To determine if a property is constrained, call **isConstrained( )**.
We can obtain the name of a property by calling **getName( )**.

EventSetDescriptor

The **EventSetDescriptor** class represents a set of Bean events.

It supports several methods that obtain the methods that a Bean uses to add or remove event listeners, and to otherwise manage events.

For example, to obtain the method used to add listeners, call **getAddListenerMethod( )**.

To obtain the method used to remove listeners, call **getRemoveListenerMethod( )**.

To obtain the type of a listener, call **getListenerType( )**.

We can obtain the name of an event set by calling **getName( )**.

MethodDescriptor

The **MethodDescriptor** class represents a Bean method.

To obtain the name of the method, call **getName( )**.

You can obtain information about the method by calling **getMethod( )**, shown here:
Method getMethod( )

An object of type **Method** that describes the method is returned.

A Bean Example

This chapter concludes with an example that illustrates various aspects of Bean programming, including introspection and using a **BeanInfo** class. It also makes use of the **Introspector**, **PropertyDescriptor**, and **EventSetDescriptor** classes. The example uses three classes. The first is a Bean called **Colors**, shown here:

```
// Colors.java
import java.awt.*;
import java.awt.event.*;
import java.io.Serializable;

public class Colors extends Canvas implements Serializable {

    transient private Color color;
    private boolean rectangular;

    public Colors() {
        addMouseListener(new MouseAdapter() {
            public void mousePressed(MouseEvent me) {
                change();
            }
        });


        addMouseListener(new MouseAdapter() {

            @Override
            public void mouseWheelMoved(MouseWheelEvent e) {
                super.mouseWheelMoved(e); //To change body of generated methods, choose Tools | 
Templates.
                change();
            }

        });
        rectangular = false;
        setSize(200, 100);
        change();
    }

    public boolean getRectangular() {
        return rectangular;
    }

    public void setRectangular(boolean flag) {
        this.rectangular = flag;
        repaint();
    }

    public void change() {
        color = randomColor();
        repaint();
```

```
    }

    private Color randomColor() {
        int r = (int) (255 * Math.random());
        int g = (int) (255 * Math.random());
        int b = (int) (255 * Math.random());
        return new Color(r, g, b);
    }

    public void paint(Graphics g) {
        Dimension d = getSize();
        int h = d.height;
        int w = d.width;
        g.setColor(color);
        if (rectangular) {
            g.fillRect(0, 0, w - 1, h - 1);
        } else {
            g.fillOval(0, 0, w - 1, h - 1);
        }
    }
}


// ColorsBeanInfo.java

import java.beans.*;

public class ColorsBeanInfo extends SimpleBeanInfo {

    public PropertyDescriptor[] getPropertyDescriptors() {
        try {
            PropertyDescriptor rectangular = new PropertyDescriptor("rectangular", Colors.class);
            PropertyDescriptor pd[] = {rectangular};
            return pd;
        } catch (Exception e) {
            System.out.println("Exception caught. " + e);
        }
        return null;
    }
}

// IntrospectorDemo.java //Show properties and events.
import java.beans.BeanInfo;
import java.beans.EventSetDescriptor;
import java.beans.Introspector;
import java.beans.PropertyDescriptor;

public class IntrospectorDemo {

    public static void main(String args[]) {
        try {
            Class c = Class.forName("Colors");
            BeanInfo beanInfo = Introspector.getBeanInfo(c);
            System.out.println("Properties:");
            PropertyDescriptor propertyDescriptor[] = beanInfo.getPropertyDescriptors();
            for (int i = 0; i < propertyDescriptor.length; i++) {
                System.out.println("\t" + propertyDescriptor[i].getName());
            }
```

```
            System.out.println("Events:");
            EventSetDescriptor eventSetDescriptor[] = beanInfo.getEventSetDescriptors();
            for (int i = 0; i < eventSetDescriptor.length; i++) {
                System.out.println("\t" + eventSetDescriptor[i].getName());
            }
        } catch (Exception e) {
            System.out.println("Exception caught. " + e);
        }
    }
}
```

Goto NetBean-> Tools-> Palette -> Add from JAR-> add "Colors" class-> Bean.

Create an New FrameColors.java-> Left side "Palette" ->see in " Beans" option.-> choose Colors Bean.- > Drag Colors to an Frame.

"A **POJO** is a **Plain Old Java Object**. These objects didn't have any super-functions and didn't inherit super-objects. They were just regular Java objects."

"When you get to know EJB in practice, you'll understand the difference. Roughly speaking, a POJO is a knife, and an EJB is a Swiss Army knife that you can also use to make phone calls."

"A data transfer object (**DTO**) is an object created to transport data. These objects usually have two requirements. They must: a) be able to store data, b) be serializable. In other words, they are used only for transferring data."

"You create an object, write the required data from the business logic into it, serialize it into JSON or XML, and send it where it needs to go. Or the other way around: a message arrives, you deserialize it into a DTO object, and extract data from it."

"An **Entity** is an object that is stored in a database. But they don't contain any business logic. You could say that this the business model's data."

"We also have the data access object (**DAO**). A DAO is used to save objects to and retrieve them from a database. The entity doesn't do this, since it doesn't have any logic, so it can't save anything anywhere."

Before going to write a JavaBean, here are some basic rules. A JavaBean should be public, should has no argument default constructor and should implement serializable interface. Keep these basic rules in mind before writing a JavaBean.

```
//Colors.java
import java.awt.*;
import java.awt.event.*;
public class Colors extends Canvas {
   transient private Color color;
   private boolean rectangular;
```

```
   public Colors() {
      addMouseListener(new MouseAdapter() {
         public void mousePressed(MouseEvent me) {
            change();
         }
      });
      rectangular = false;
      setSize(200, 100);
      change();
   }
   public boolean getRectangular() {
      return rectangular;
   }
   public void setRectangular(boolean flag) {
      this.rectangular = flag;
      repaint();
   }
   public void change() {
      color = randomColor();
      repaint();
   }
   private Color randomColor() {
      int r = (int)(255*Math.random());
      int g = (int)(255*Math.random());
      int b = (int)(255*Math.random());
      return new Color(r, g, b);
   }
   public void paint(Graphics g) {
      Dimension d = getSize();
      int h = d.height;
      int w = d.width;
      g.setColor(color);
      if(rectangular) {
   g.fillRect(0, 0, w-1, h-1);
   }
   else {
   g.fillOval(0, 0, w-1, h-1);
   }
   }
}
```