Unit – 3 Java Beans [7 Hrs.]

Introduction to Java Beans:

A JavaBean is a software component that has been designed to be reusable in a variety of environments. *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.

Syntax for setter methods:

1. It should be public in nature.
2. The return-type should be void.
3. The setter method should be prefixed with set.
4. It should take some argument i.e. it should not be no-arg method.

Syntax for getter methods:

1. It should be public in nature.
2. The return-type should not be void i.e. according to our requirement we have to give return-type.
3. The getter method should be prefixed with get.
4. It should not take any argument.

For Boolean properties getter method name can be prefixed with either "get" or "is". But recommended to use "is".

Example of Simple Java Bean:

Creating a Java Bean:

StudentBean.java

```
import java.io.*;
public class StudentBean implements Serializable{
   private int id;
   private String name;

   public void setId(int id) {
        this.id=id;
   }
```

```
    public void setName(String name) {
          this.name=name;
    }

    public int getId() {
          return id;
    }

    public String getName() {
          return name;
    }
}
```

Accessing Java Bean

Test.java

```
public class Test {
    public static void main(String[] args) {
          //accessing info from bean
          StudentBean sb=new StudentBean();
          sb.setId(101);
          sb.setName("Raaju Poudel");
          System.out.println("Id: "+sb.getId());
          System.out.println("Name: "+sb.getName());
    }
}
```

Output:

```
Id: 101
Name: Raaju Poudel
```

Advantages of Java Beans:

A software component architecture provides standard mechanisms to deal with software building blocks. The following list enumerates some of the specific benefits that Java technology provides for a component developer:

- A Bean obtains all the benefits of Java's "write-once, run-anywhere" paradigm.
- The properties, events, and methods of a Bean that are exposed to an application builder tool can be controlled.
- A Bean may be designed to operate correctly in different locales, which makes it useful in global markets.
- Auxiliary software can be provided to help a person configure a Bean. This software is only needed when the design-time parameters for that component are being set. It does not need to be included in the run-time environment.
- The configuration settings of a Bean can be saved in persistent storage and restored at a later time.
- A Bean may register to receive events from other objects and can generate events that are sent to other objects.

Introspection

Introspection is the process of analyzing a Bean to determine its capabilities. This is an essential feature of the Java Beans API, because it allows an application builder tool to present information about a component to a software designer. 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:

```
public class Box {

    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;
    }
}
```

Boolean Properties:

A Boolean property has a value of true or false. It can be identified by the following design patterns, where N is the name of the property:

```
public boolean isN( );
public boolean getN( );
public void setN(boolean value);
```

Either the first or second pattern can be used to retrieve the value of a Boolean property. However, if a class has both of these methods, the first pattern is used.

The following listing shows a class that has one Boolean property:

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

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. 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
                    TooManyListeners;
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) {
…
}
```

Using the BeanInfo Interface

BeanInfo is an interface implemented by a class that provides explicit information about a Bean. **It is used to describe one or more feature sets of a Bean, including its properties, methods, and events.**

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.

Bound and Constrained Properties

A Bean that has a *bound* property generates an event when the property is changed. The event is of type **PropertyChangeEvent** and is sent to objects that previously registered an interest in receiving such notifications. A class that handles this event must implement the **PropertyChangeListener** interface.

A Bean that has a *constrained* property generates an event when an attempt is made to change its value. It also generates an event of type **PropertyChangeEvent**. It too is sent to objects that previously registered an interest in receiving such notifications. However, those other objects have the ability to veto the proposed change by throwing a **PropertyVetoException**. This capability allows a Bean to operate differently according to its run-time environment. A class that handles this event must implement the **VetoableChangeListener** interface.

Persistence

***Persistence* is the ability to save the current state of a Bean**, including the values of a Bean's properties and instance variables, to nonvolatile storage and to retrieve them at a later time. The object serialization capabilities provided by the Java class libraries are used to provide persistence for Beans.

The easiest way to serialize a Bean is to have it implement the **java.io.Serializable** interface, which is simply a marker interface. Implementing **java.io.Serializable** makes serialization automatic. Your Bean need take no other action. Automatic serialization can also be inherited. Therefore, if any superclass of a Bean implements **java.io.Serializable**, then automatic serialization is obtained.

When using automatic serialization, you can selectively prevent a field from being saved through the use of the **transient** keyword. Thus, data members of a Bean specified as **transient** will not be serialized.

If a Bean does not implement **java.io.Serializable**, you must provide serialization yourself, such as by implementing **java.io.Externalizable**. Otherwise, containers cannot save the configuration of your component.

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.

The JavaBeans API

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

Classes in java.bean package:

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	Supports 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 **PropertyDescriptor**, **EventSetDescriptor**, and **MethodDescriptor** classes, among others.

IndexedPropertyChangeEvent	A subclass of **PropertyChangeEvent** that represents a change to an indexed property.
IndexedPropertyDescriptor	Instances of this class describe an indexed property 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.
PropertyChangeEvent	This event is generated when bound or constrained properties are changed. It is sent to objects that registered an interest in these events and that implement either the **PropertyChangeListener** or **VetoableChangeListener** interfaces.
PropertyChangeListenerProxy	Extends **EventListenerProxy** and implements **PropertyChangeListener**.
PropertyChangeSupport	Beans that support bound properties can use this class to notify **PropertyChangeListener** objects.
PropertyDescriptor	Instances of this class describe a property of a Bean.
PropertyEditorManager	This class locates a **PropertyEditor** object for a given type.
PropertyEditorSupport	This class provides functionality that can be used when writing property editors.
PropertyVetoException	An exception of this type is generated if a change to a constrained property 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**.
VetoableChangeSupport	Beans that support constrained properties 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.

Interface in java.beans package:

Interface	Description
AppletInitializer	Methods in this interface are used to initialize Beans that are also applets. (Deprecated by JDK 9.)
BeanInfo	This interface allows a designer to specify information about the properties, events, and methods of a Bean.
Customizer	This interface allows a designer to provide a graphical user interface through which a Bean may be configured.
DesignMode	Methods in this interface determine if a Bean is executing in design mode.
ExceptionListener	A method in this interface is invoked when an exception has occurred.
PropertyChangeListener	A method in this interface is invoked when a bound property is changed.
PropertyEditor	Objects that implement this interface allow designers to change and display property values.
VetoableChangeListener	A method in this interface is invoked when a constrained property is changed.
Visibility	Methods in this interface allow a Bean to execute in environments where a graphical user interface is not available.

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,

- You can determine if a property is bound by calling **isBound( )**.
- To determine if a property is constrained, call **isConstrained( )**. You 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( )**. You 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( )**,

Example is shown below,

```
Method getMethod()
```

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

Using Java Beans

Example – 1 (Example using PropertyDescriptor)

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

//create a property set of Fruit class
class Fruit implements Serializable{
	private String name;
	private double price;

	public void setName(String name) {
		this.name=name;
	}

	public String getName() {
		return name;
	}

	public void setPrice(double price) {
		this.price=price;
	}

	public double getPrice() {
		return price;
	}
}
```

```
//Accessing properties using Property Descriptor
public class BeanExample {
   public static void main(String[] args) throws Exception {

         BeanInfo beanInfo=Introspector.getBeanInfo(Fruit.class);
         PropertyDescriptor pd[]=beanInfo.getPropertyDescriptors();

         System.out.println("Properties:");
         for(int i=0;i<pd.length;i++) {
               System.out.println("\t"+pd[i].getName());
         }

   }
}
```

Output:

```
Properties:
      class
      name
      price
```

Note: All properties are displayed including super class.

Example – 2 (Example using EventSetDescriptor)

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

//create a property set of Fruit class
class Fruit extends Canvas implements Serializable{
      //declaring mouse event
      public Fruit() {
            addMouseListener(new MouseAdapter() {
                  public void MousePressed(MouseEvent me) {
                        //your stuffs on mouse pressed
                  }
            });
      }
}

//Accessing properties using Property Descriptor
public class BeanExample {
   public static void main(String[] args) throws Exception {
         BeanInfo beanInfo=Introspector.getBeanInfo(Fruit.class);
         EventSetDescriptor ed[]=beanInfo.getEventSetDescriptors();
         System.out.println("Events:");
         for(int i=0;i<ed.length;i++) {
               System.out.println("\t"+ed[i].getName());
         }
   }
}
```

Output:

```
Events:
      hierarchyBounds
      mouse
      component
      hierarchy
      inputMethod
      mouseMotion
      focus
      propertyChange
      mouseWheel
      key
```

Note: All events are displayed including super class Canvas.

Example – 3 (Example using MethodDescriptor)

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

//create a property set of Fruit class
class Fruit implements Serializable{
      private String name;
      private double price;

      public void setName(String name) {
            this.name=name;
      }

      public String getName() {
            return name;
      }

      public void setPrice(double price) {
            this.price=price;
      }

      public double getPrice() {
            return price;
      }
}

//Accessing properties using Property Descriptor
public class BeanExample {
   public static void main(String[] args) throws Exception {
         BeanInfo beanInfo=Introspector.getBeanInfo(Fruit.class);
         MethodDescriptor md[]=beanInfo.getMethodDescriptors();
         System.out.println("Methods:");
         for(int i=0;i<md.length;i++) {
               System.out.println("\t"+md[i].getName());
         }
   }
}
```

Output:

```
Methods:
      getClass
      getName
      setPrice
      setName
      wait
      notifyAll
      notify
      wait
      hashCode
      getPrice
      wait
      equals
      toString
```

Example – 4 (Example by extending SimpleBeanInfo class)

Colors.java

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

public class Colors implements Serializable {
    private int id;
      private String name;

    public void setId(int id) {
      this.id=id;
    }

    public int getId() {
      return id;
    }

    public void setName(String name) {
      this.name=name;
    }

    public String getName() {
      return name;
    }
}
```

ColorsBeanInfo.java

```
import java.beans.*;
public class ColorsBeanInfo extends SimpleBeanInfo{
   public PropertyDescriptor[] getPropertyDescriptors() {
         try {
               PropertyDescriptor name=new PropertyDescriptor
                           ("name",Colors.class);
               PropertyDescriptor pd[]= {name};
               return pd;
         }catch(Exception ex) {
               System.out.println(ex.toString());
         }
         return null;
   }
}
```

IntroSpectorDemo.java

```
import java.awt.*;
import java.beans.*;
public class IntroSpectorDemo {
   public static void main(String[] args) throws Exception {
         BeanInfo beanInfo=Introspector.getBeanInfo(Colors.class);
         System.out.println("Properties:");
         PropertyDescriptor pd[]=beanInfo.getPropertyDescriptors();
         for(int i=0;i<pd.length;i++) {
               System.out.println("\t"+pd[i].getName());
         }
   }
}
```

Output:

```
Properties:
      name
```

Using Java Beans in JSP

<jsp:useBean>

The jsp:useBean tag is utilized to start up an object of JavaBean or it can re-utilize the existing java bean object. The primary motivation behind jsp:useBean tag is to interface with bean objects from a specific JSP page. In this tag, it is consistently suggestible to give either application or meeting degree to the extension characteristic worth. At the point when the compartment experiences this label then the holder will get class characteristic worth for example completely qualified name of Bean class then the holder will perceive Bean .class record and perform Bean class stacking and launch. Subsequent to making the Bean object compartment will allot Bean object reference to the variable indicated as an incentive to id property. In the wake of getting Bean object reference holder will store Bean object in an extension indicated as an incentive to scope trait.

Syntax: <jsp:useBean id=”—” class”—” type”—” scope=”—”/>

Attributes:

1. **Id**: It will take a variable to manage generated Bean object reference.
2. **Class**: This attribute will take the fully qualified name of the Bean class.
3. **Type**: It will take the fully qualified name of the Bean class to define the type of variable in order to manage the Bean object reference.
4. **Scope**: It will take either of the JSP scopes to the Bean object.

<jsp:getProperty>

The jsp:getProperty tag is utilized to get indicated property from the JavaBean object. The principle reason for <jsp:getProperty> tag is to execute a getter strategy to get an incentive from the Bean object.

Syntax: <jsp:getProperty name="—" property="—"/>

Attributes:

1. **Name**: It will take a variable that is the same as the id attribute value in <jsp:useBean> tag.
2. **Property**: It will take a particular property to execute the respective getter method.

<jsp:setProperty>

The jsp:setProperty tag is utilized to set a property in the JavaBean object. The principle motivation behind jsp:setProperty tag is to execute a specific setter technique to set an incentive to a specific Bean property.

Syntax: <jsp:setProperty name="—" property="—" value="—"/>

Attributes:

1. **Name**: It will take a variable that is the same as the id attribute value in <jsp:useBean> tag.
2. **Property**: It will take a property name in order to access the respective setter method.
3. **Value**: It will take a value to pass as a parameter to the respective setter method.

JSP JavaBeans Example

StudentBean.java

```
package mypack;
public class StudentBean {
          private String address;
          public void setAddress(String address) {
            this.address=address;
          }

          public String getAddress() {
            return address;
          }
}
```

index.jsp

```
<html>
<head>
<body>
    <h2>Using Java Bean in JSP</h2>
    <jsp:useBean id="student" class="mypack.StudentBean" />

    <jsp:setProperty name="student" property="address"
                value="Birtamod,Jhapa"/>
    <b>Address is:</b>
    <jsp:getProperty name="student" property="address"/>

</body>
</html>
```

Output: