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 obtaining an inventory value, or a complex function, such as forecasting the performance of a stock portfolio.

Advantages of Java Beans

- 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 another application can be controlled.
- Auxiliary software can be provided to help 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

- 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 method**, an additional class that extends the **BeanInfo interface is provided that** explicitly supplies this information.

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 three types of properties: simple, indexed and boolean properties*

Simple Properties

- Asimple 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)*
- Aread/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.
- An example: 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;
}
}
```

- Example 2

```
 public class student {
           private String name;
         public student()  {    }

       public void setName (String varname) {
                       name = varname;
              }
           public String getName(){
                         return name;
                  }
}
```

```
class studentDemo {
  public static void main(String args[]) {
     student obj = new student();
     obj.setName(“amit”);
     String st1 = obj.getName();
   }
}
```

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[ ]);*

```
public class Piechart {
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);
}
}
```

Boolean Properties

- A Boolean property has a value of true or false. It can be identified the following design patterns

```
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 mathods, the first pattern is used.

- Following listing shows a class that has one boolean property

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

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 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 is 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) {
  ...
  }
  ```

Using the BeanInfo Interface

- 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 methods **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** ***bnameBeanInfo,*** 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,**
- Implementing **java.io.Serializable makes serialization** automatic.
- There is one important restriction: any class that implements **java.io.Serializable must supply a parameterless constructor.**

- 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

- ABean 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.

Java Beans API

- The Java Beans functionality is provided by a set of classes and interfaces in the **java.beans** package.
- (Refer Book for list interfaces and classes At page no 852. Book: Java The Complete Reference)

Some classes defined in Java.beans package

- **Introspector**
- The **Introspector class provides several static methods that support introspection, 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 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 property by calling **getName( ).**

- **EventSetDescriptor**
- The **EventSetDescriptor class represents a Bean event. 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 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

- See program in MS word file.