Unit 3 :
Java Beans

Java Beans

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

- ❖ It is a java class that should follow **following conventions**:

1. Must implement Serializable.
2. It should have a public no-arg constructor.
3. All properties in java bean must be private with public getters and setter methods.

Why use JavaBean?

- ❖ According to Java white paper, it is a reusable software component. A bean encapsulates many objects into one object so that we can access this object from multiple places. Moreover, it provides easy maintenance.

Advantages of JavaBean:

- ❖ The JavaBean properties and methods can be exposed to another application.
- ❖ It provides an easiness to reuse the software components.
- ❖ 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.
- ❖ A Bean obtains all the benefits of Java's "write-once, run-anywhere" paradigm.

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.

Syntax for setter methods:

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

Syntax for getter methods:

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

Key Bean Concept (Features):

- JavaBean is a complete component model. It supports the standard component architecture features of properties, events, method and persistence.

- The following list briefly describes Key bean concepts :

1. Introspection:

- Introspection is the automatic process of analyzing a bean's design patterns to reveal the bean's properties, events, and methods. This process controls the publishing and discovery of bean operations and properties.
- It can be done with the help of Introspector class.

The way in which introspection is implemented provides great advantages, including:

***Portability* -** Everything is done in the Java platform, so you can write components once, reuse them everywhere. There are no extra specification files that need to be maintained independently from your component code. There are no platform-specific issues to contend with. Your component is not tied to one component model or one proprietary platform. You get all the advantages of the evolving Java APIs, while maintaining the portability of your components.

Reuse - By following the JavaBeans design conventions, implementing the appropriate interfaces, and extending the appropriate classes, you provide your component with reuse potential that possibly exceeds your expectations.

Introspection:

Syntax :

BeanInfo beaninfo = Introspector.*getBeanInfo*(Sample.**class**,Object.**class**);
PropertyDescriptor[] prop = beaninfo.getPropertyDescriptors();

```
package com.texas.javaBeans;
import java.beans.BeanInfo;
import java.beans.IntrospectionException;
import java.beans.Introspector;
import java.beans.PropertyDescriptor;
import java.io.Serializable;
import java.util.Date;

public class StudentIntospector implements Serializable{
		private static final long serialVersionUID = 1L;
		private String firstName;
		private String middleName;
		private String lastName;
		private int rollNo;
		private Date dob;
		public StudentIntospector() {
		}
		public void setFirstName(String firstName) {
				this.firstName = firstName;
		}
		public String getFirstName() {
				return firstName;
		}
		public void setRollNo(int rollNo) {
				this.rollNo = rollNo;
		}
		public int getRollNo() {
				return rollNo;
		}
		public void setDob(Date dob) {
				this.dob = dob;
		}
		public Date getDob() {
				return dob;
		}
public static void main(String[] args) {
		try {
			BeanInfo beanInfo =  Introspector.getBeanInfo(StudentIntospector.class);
			PropertyDescriptor[] propBeanInfo =  beanInfo.getPropertyDescriptors();
			for(PropertyDescriptor prop : propBeanInfo) {
				System.out.println(prop.getName());
				System.out.println("Get Method Name : "+prop.getReadMethod());
				System.out.println("Set Method Name : "+prop.getWriteMethod());
			}
		} catch (IntrospectionException e) {
			e.printStackTrace();
		}
	}
}
```

2. 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. They are set through a setter method and can be obtained by a getter method.

Types of properties :

a. Simple Properties
b. Boolean Properties
c. Indexed Properties
d. Bound Properties
e. Constrained Properties

A. <u>Simple Properties :</u>

- ❖ Simple property are basic, independent, individual properties like firstName, lastName, height, width etc.
- ❖ The accessor and mutator method should follow standard naming conventions.

e.g.

```
private String firstName;

public void setFirstName(String firstName) {
this.firstName = firstName;
}


public String getFirstName() {
return firstName;
}
```

B. <u>Boolean Properties :</u>

- ❖ They are simple properties.
- ❖ The accessor and mutator method should follow standard naming conventions.
- ❖ The getter methods follow an optional design pattern.

e.g.

```
private boolean connect;

public boolean isConnect() {
return connect;
}


public void setConnect(boolean connect) {
this.connect = connect;
}
```

C. Indexed Properties :

❖ It is a property that can take an array of values.
❖ It supports a range of values instead of single value.

e.g.

```
private int[] rollNo;

public int[] getRollNo() {
return rollNo;
}

public void setRollNo(int[] rollNo) {
this.rollNo = rollNo;
}
```

D. Bound Properties :

❖ Sometimes when a Bean property changes, another object may want to be notified of the change, and react to the change. Whenever a *bound property* changes, notification of the change is sent to interested listeners.

This has two implications:

a. The bean class includes addPropertyChangeListener() and removePropertyChangeListener() methods for managing the bean's listeners.
b. When a bound property is changed, the bean sends a PropertyChangeEvent to its registered listeners.

To implement the bound property, take the following steps :

1. **Import the java.beans package.**

2. **Instantiate a PropertyChangeSupport object** – *This object maintains the property change listener list and fires property change events.*

e.g.
Private PropertyChangeSupport pcs = new PropertyChangeSupport(this);

3. Implement methods to maintain the property change listener list.

e.g.

```
public void addPropertyChangeListener(PropertyChangeListener propertyChangeListener) {
        pcs.addPropertyChangeListener(propertyChangeListener);
}
public void removePropertyChangeListener(PropertyChangeListener propertyChangeListener) {
        pcs.removePropertyChangeListener(propertyChangeListener);
}
```

4. Modify a property's setter method to fire a property change event when the property is changed.

```
public void setPropertyName(PropertyType pt) {
    pcs.firePropertyChange(”Demo", old, new);
}
```

*The **firePropertyChange** method bundles its parameters into a **PropertyChangeEvent** object, and calls **propertyChange**(**PropertyChangeEvent** pce) on each registered listener.*
Also, property change events are fired after the property has changed.

5. Implement the PropertyChangeListener interface.

6. Implement the propertyChange method in the listener.

This method needs to contain the code that handles what you need to do when the listener receives property change event.

Source bean:

```
package com.texas.boundproperty;
import java.io.Serializable;
import java.beans.*;

public class Parent implements Serializable {

        private static final long serialVersionUID = 1L;
        private String firstName = "Shankar";
        private String lastName = "Dahal";
        private PropertyChangeSupport pcs;

        public Parent() {
                pcs = new PropertyChangeSupport(this);
        }
        public void setFirstName(String firstName) {
                this.firstName = firstName;
        }
        public String getFirstName() {
                return firstName;
        }
        public void setLastName(String newLastName) {
                String oldLastName = lastName;
                pcs.firePropertyChange("LastName", oldLastName, newLastName);
                this.lastName = newLastName;
        }
        public String getLastName() {
                return lastName;
        }
        public void addPropertyChangeListener(PropertyChangeListener pcl) {
                pcs.addPropertyChangeListener(pcl);

        }
        public void removePropertyChangeListener(PropertyChangeListener pcl) {
                pcs.removePropertyChangeListener(pcl);
        }
}
```

```
public static void main(String[] args) {

Parent boundPropertyDemo = new Parent();
boundPropertyDemo.addPropertyChangeListener(new PropertyChangeListener() {

public void propertyChange(PropertyChangeEvent evt) {

System.out.println("Old : "+evt.getOldValue() + " New Value : "+evt.getNewValue());
}
});

boundPropertyDemo. setLastName("Dahal");
boundPropertyDemo. setLastName("Shrestha");
String lastName  = boundPropertyDemo. getLastName();

System.out.println("LastName : "+ lastName);

}
```

E. Constrained Properties :

- A bean property is *constrained* if the bean supports the VetoableChangeListener and PropertyChangeEvent classes, and if the set method for this property throws a PropertyVetoException .

In a simple :

- 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.

The basic idea for implementing a constrained property is as follows:

i. Store the old value of source bean's property if it is to be vetoed.
ii. Ask listeners (vetoers) of the new proposed value.
iii. Vetoers are authoritative to process this new value. They disallow by throwing an exception. Note that no exception is thrown if a vetoer gives the permission.
iv. If no listener vetoes the change, i.e., no exception is thrown, the property is set to the new value; optionally notify "PropertyChange" listeners, if any.

Three objects are involved in this process.
i. The source bean containing one or more constrained property
ii. A listener object that accepts or rejects changes proposed by the source bean
iii. A PropertyChangeEvent object encapsulating the property change information

To implement the constrained property, take the following steps :

1. **Import the java.beans package.**

2. **Instantiate a VetoableChangeSupport object**

e.g.

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

3. **Implement methods to maintain the vetoable change listener list.**

e.g.

```
public void addVetoableChangeListener(VetoableChangeListener vetoableChangeListener) {
  vcs.addVetoableChangeListener(vetoableChangeListener);
}
public void removeVetoableChangeListener(VetoableChangeListener vetoableChangeListener) {
  vcs.removeVetoableChangeListener(vetoableChangeListener);
}
```

4. **The accessor methods for a constrained property are defined in the same way as those for simple properties, with the addition that the setXXX method throws a PropertyVetoException exception.**

```
public void setPropertyName(PropertyType pt) throws PropertyVetoException {
        vcs.fireVetoableChange(”X", old, new);
}
```

5. Implement the VetoableChangeListener interface.

6. Implement the vetoableChange method in the listener.

This method needs to contain the code that handles what you need to do when the listener receives veto change event.

e.g.

```
public void vetoableChange(PropertyChangeEvent evt) throws PropertyVetoException {
        throw new PropertyVetoException("Message", evt);
}
```

Example : Source Bean:

```
package com.texas.constrained;
import java.io.Serializable;
import java.beans.*;

public class Broker implements Serializable {
        private static final long serialVersionUID = 1L;
        private int price = 1200;
        private VetoableChangeSupport vcs;

        public Broker() {
                vcs = new VetoableChangeSupport(this);
        }
        public void setPrice(int newPrice) throws PropertyVetoException {
                int oldPrice = price;
                this.price = newPrice;
                vcs.fireVetoableChange("Price", oldPrice, newPrice);
        }
        public int getPrice() {
                return price;
        }
        public void addVetoableChangeListener(VetoableChangeListener vcl) {
                vcs.addVetoableChangeListener(vcl);
        }
        public void removeVetoableChangeListener(VetoableChangeListener vcl) {
                vcs.removeVetoableChangeListener(vcl);
        }
}
```

The ShareHolder is the listener bean. It decides whether the Broker can sell shares at the specified price. It allows the Broker selling shares if the proposed price is greater than or equal to to 1210. Otherwise, it disallows, by throwing the PropertyVetoException object.

```
package com.texas.constrained;
import java.beans.PropertyChangeEvent;
import java.beans.PropertyVetoException;
import java.beans.VetoableChangeListener;

public class ShareHolder implements VetoableChangeListener {
        public void vetoableChange(PropertyChangeEvent evt) throws PropertyVetoException {
                if (evt.getPropertyName().equals("Price")) {
                        int newPrice = (int) evt.getNewValue();
                        if (!(newPrice >= 1210)) {
                                throw new PropertyVetoException("Not interested.", evt);
                        }
                }
        }
}
```

For demonstration purposes, we have created the following Java application program.

```
package com.texas.constrained;
import java.beans.PropertyVetoException;

public class MainConstrainedDemo {
        public static void main(String[] args) {
                Broker broker = new Broker();
                ShareHolder shareHolder = new ShareHolder();
                broker.addVetoableChangeListener(shareHolder);
                System.out.println("Before Change ");
                System.out.println("Old price : " + broker.getPrice());
                try {
                        broker.setPrice(1209);
                        System.out.println("New price : " + broker.getPrice());
                } catch (PropertyVetoException e) {
                        System.out.println("Message : " + e.getMessage());
                }
        }
}
```

3. Persistence:

- Persistence is a procedure to save a bean in non-volatile storage such as a file. The bean can be reconstructed and used later. The important point is that persistence allows bean developers to save the current state of the bean and retrieve the same at some later point of time.

- The mechanism that makes persistence possible is called *serialization*. Object serialization means converting an object into a data stream and writing it to storage.

Example:

Factorial bean.

```
package com.texas.javabeansdemo;
import java.io.Serializable;

public class Factorial implements Serializable {
		private static final long serialVersionUID = 1L;
		private int n;
		protected int factorial;
		public Factorial() {
		}
		public int getN() {
				return n;
		}
		public void setN(int n) {
				this.n = n;
				int fact = 1;
				for (int i = 1; i <= n; i++) {
						fact = fact * i;
				}
				factorial = fact;
		}
		public int getFactorial() {
				return factorial;
		}
}
```

The following code shows how to **save** the state of a bean.

```
Factorial factorial = new Factorial();
for (int i = 1; i <= 10; i++) {
		factorial.setN(i);
		System.out.println("Factorial of " + i + " is " + factorial.getFactorial());
		if (i > Math.random() * 10) {
				break;
		}
}

// Save
FileOutputStream fileOutputStream = new FileOutputStream("FactorialBean.dat");
ObjectOutputStream objectOutputStream = new ObjectOutputStream(fileOutputStream);
objectOutputStream.writeObject(factorial);
objectOutputStream.close();
```

The following code shows how to **retrieve** the state of a bean.

```
FileInputStream fileInputStream = new FileInputStream("FactorialBean.dat");
ObjectInputStream objectInputStream = new ObjectInputStream(fileInputStream);
Factorial factorial2 = (Factorial) objectInputStream.readObject();
int n = factorial2.getN();
System.out.println(n);
objectInputStream.close();
```

5. Customizer

Bean customization allows us to customize the appearance and behaviour of a bean at design time, within a bean-compliant builder tool.

Typically, there are two ways to customize a Bean:

1. By using a property editor
Each Bean property has its own property editor. A builder tool usually displays a Bean's property editors in a property sheet. A property editor is associated with, and edits a particular property type.

2. By using customizers
Customizers give you complete GUI control over Bean customization. Customizers are used where property editors are not practical or applicable.

Customizer interface look like this:

```
public interface Customizer {

  void setObject(Object bean);
  void addPropertyChangeListener(PropertyChangeListener listener);
  void removePropertyChangeListener(PropertyChangeListener listener);
}
```

Design Pattern

- ❖ Property design patterns are used to identify the properties of a Bean. Property design patterns are closely related to accessor and mutator methods.

Types of property design patterns

a. Simple property design patterns
b. Boolean property design patterns
c. Indexed property design patterns

Multicast Event Design Patterns

- ❖ Beans that support multiple event listeners are multicast event sources. The automatic JavaBeans introspection facility uses a pair of special event registration methods for Beans that broadcast to multiple event listeners: one method allows interested parties to be added as listeners, and the other allows listeners to be removed. Following are the design patterns governing these event registration methods:

```
public void addEventListenerType (EventListenerType x);
public void removeEventListenerType (EventListenerType x);
```

Simple property design pattern

- ❖ Simple properties in JavaBeans consist of all single-valued properties, which include all built-in Java data types as well as classes and interfaces.
- ❖ For example, properties of type int, long, float, Color, Font, and boolean are all considered simple properties.
- ❖ The design patterns for simple property accessor and mutator methods as follow:

```
public PropertyType getPropertyName();
public void setPropertyName(PropertyType x);
```

Boolean property design pattern

- ❖ Boolean properties have an optional design pattern for the getter method that can be used to make it more clear that the property is boolean.
- ❖ The design pattern for the boolean getter method follows:

```
public boolean isPropertyName();
```

Following is an example of a pair of accessor and mutator methods for a boolean property named **visible**:

```
public boolean isVisible();
public void setVisible(boolean v);
```

Indexed property design patterns

- An indexed property is a property representing an array of values.
- Indexed properties require slightly different accessor methods, it only makes sense that their design patterns differ a little from those of other properties.
- Following are the design patterns for indexed properties:

```
public PropertyType getPropertyName(int index);
public void setPropertyName(int index, PropertyType x);
public PropertyType[] getPropertyName();
public void setPropertyName(PropertyType[] x);
```

The first pair of design patterns defines accessor methods for getting and setting individual elements in an indexed property. The second pair of patterns defines accessor methods for getting and setting the entire property array as a whole.

BeanInfo Interface

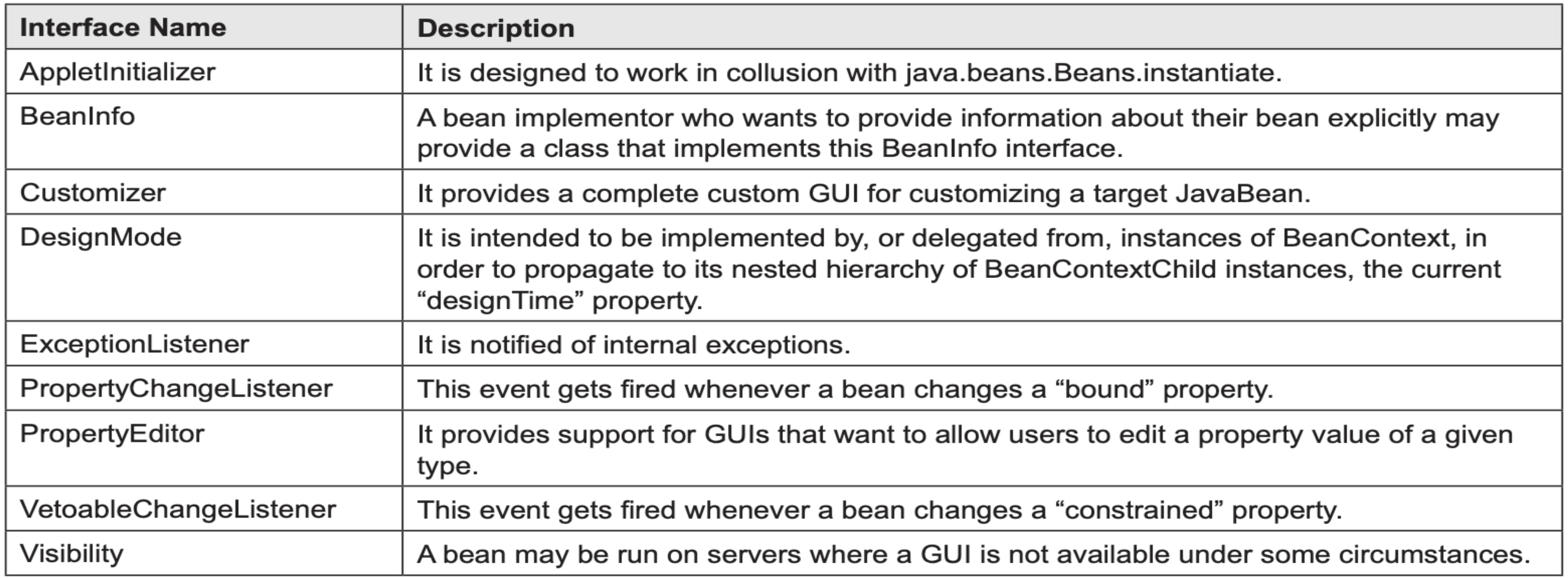
- ❖ A bean builder typically uses the introspection process to discover features (such as properties, methods, and events).
- ❖ In this case, the bean builder exposes all the features to the outside world.
- ❖ BeanInfo is an interface implemented by a class that provides explicit information about a Bean. It is used to describe one or more feature sets of a Bean, including its properties, methods, and events.
- ❖ The features of a bean are exposed using a separate special class. This class must implement the BeanInfo interface. The BeanInfo interface defines several methods that can be used to inspect properties, methods, and events of a bean. The following is the declaration of the BeanInfo interface:

```
public interface BeanInfo {
        BeanInfo[] getAdditionalBeanInfo();
        BeanDescriptor getBeanDescriptor();
        int getDefaultEventIndex();
        int getDefaultPropertyIndex();
        EventSetDescriptor[] getEventSetDescriptors();
        MethodDescriptor[] getMethodDescriptors();
        PropertyDescriptor[] getPropertyDescriptors();
}
```

JavaBean API

- ❖ JavaBean API provides a set of related interfaces and classes necessary to design and develop beans in a separate package, java.beans. Not all the classes and interfaces are used all the time to develop a bean.
- ❖ For example, the event classes are used by beans that fire property and vetoable change events. However, most of the classes in this package are used by a bean builder/editor.
- ❖ In particular, these classes and interfaces help the bean builder/editor to create user interfaces that the user can use to customize their beans.

- ❖ Below tables show the list of interfaces, classes, and exceptions, which are used to develop beans.



Interface Name	Description
AppletInitializer	It is designed to work in collusion with java.beans.Beans.instantiate.
BeanInfo	A bean implementor who wants to provide information about their bean explicitly may provide a class that implements this BeanInfo interface.
Customizer	It provides a complete custom GUI for customizing a target JavaBean.
DesignMode	It is intended to be implemented by, or delegated from, instances of BeanContext, in order to propagate to its nested hierarchy of BeanContextChild instances, the current “designTime” property.
ExceptionListener	It is notified of internal exceptions.
PropertyChangeListener	This event gets fired whenever a bean changes a “bound” property.
PropertyEditor	It provides support for GUIs that want to allow users to edit a property value of a given type.
VetoableChangeListener	This event gets fired whenever a bean changes a “constrained” property.
Visibility	A bean may be run on servers where a GUI is not available under some circumstances.

Class Name	Description
BeanDescriptor	It provides global information about a bean, including Java class and displayName.
Beans	It provides some general purpose methods to control beans.
DefaultPersistenceDelegate	It is an implementation of the abstract PersistenceDelegate class and is the delegate used by default for classes about which no information is available.
Encoder	A class that is used to create streams or files which encode the state of a JavaBean collection in terms of its public APIs.
EventHandler	It provides support for dynamically generating event listeners, whose methods execute a statement involving an incoming event object and a target object.
EventSetDescriptor	It describes a group of events that a specified Javabean fires.
Expression	It represents a primitive expression where a single method is applied to a target and a set of arguments to return a result.
FeatureDescriptor	It is the common baseclass for PropertyDescriptor, EventSetDescriptor, and MethodDescriptor, etc.
IndexedPropertyChangeEvent	This event gets delivered whenever a component that conforms to the JavaBeans specification (a "bean") changes a bound indexed property.
IndexedPropertyDescriptor	It describes a property that acts as an array and has an indexed read and/or indexed write method to access specific elements of the array.
Introspector	This class provides a standard way for buider tools to learn about the properties, methods, and events that a target JavaBean supports.
MethodDescriptor	It describes a particular method that a JavaBean supports for external access.
ParameterDescriptor	It allows bean implementors to provide additional information for each parameter.
PersistenceDelegate	It 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	This event is delivered when a "bound" or "constrained" property is changed.
PropertyChangeListenerProxy	It extends the EventListenerProxy to add a named PropertyChangeListener.
PropertyChangeSupport	It is a utility class that can be used by beans that support bound properties.
PropertyDescriptor	It describes a property that a JavaBean exports through accessor methods.
PropertyEditorManager	It is used to locate a property editor for any given type name.
PropertyEditorSupport	It helps to 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.
VetoableChangeListenerProxy	It extends the EventListenerProxy specifically for associating a VetoableChangeListener with a "constrained" property.
VetoableChangeSupport	It is a utility class that is 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 ObjectInputStream.
XMLEncoder	The XMLEncoder class is a complementary alternative to ObjectOutputStream and can be used to generate a textual representation of a JavaBean in the same way that the ObjectOutputStream can be used to create binary representation of Serializable objects.

Exception Name	Description
IntrospectionException	It is thrown when an exception happens during Introspection.
PropertyVetoException	It is thrown when a proposed change to a property represents an unacceptable value.