**WHAT IS JAVABEAN?**

**JavaBeans are software component models. A JavaBean is a general-purpose component model. A Java Bean is a reusable software component that can be visually manipulated in builder tools. Their primary goal of a JavaBean is WORA (Write Once Run Anywhere).**

**JavaBeans should adhere to portability, reusability and interoperability.**

**JavaBeans will look a plain Java class written with getters and setters methods**

**What differentiates Beans from typical Java classes is introspection. Tools that recognize predefined patterns in method signatures and class definitions can "look inside" a Bean to determine its properties and behaviour.**

**In effect, Beans publish their attributes and behaviours through special method signature patterns that are recognised by beans-aware application construction tools.**

**JAVABEAN CONCEPTS**

**JavaBeans is a complete component model. It supports the standard component architecture features of properties, events, methods, and persistence. In addition, JavaBeans provides support for introspection (to allow automatic analysis of a JavaBeans component) and customisation (to make it easy to configure a JavaBeans component).**

**Typical unifying features that distinguish a Bean are:**

**•Introspection**

**Builder tools discover a Bean’s features (ie its properties, methods, and events) by a process known as INTROSPECTION. Beans supports introspection in two ways:**

**1)Low Level Introspection (Reflection) + Intermediate Level Introspection (Design Pattern):**

**Low Level Introspection is accomplished using java.lang.reflectpackage API. This API allows Java Objects to discover information about public fields, constructors, methods and events of loaded classes during program execution i.e., at Run-Time.**

**Intermediate Level Introspection (Design Pattern)is accomplished using Design Patterns. Design Patterns are bean features naming conventions to which one has to adhere while writing code for Beans. java.beans.Introspector class examines Beans for these design patterns to discover Bean features. The Introspector class relies on the core reflection API. There are two types of methods namely, accessor methods and interface methods.**

**Accessor methods are used on properties and are of two sub-types (namely getter methods and setters methods).**

**Interface methods are often used to support event handling.**

**2) Higest Level or Explicit Introspection (BeanInfo):**

**It is accomplished by explicitly providing property, method, and event information with a related Bean Information Class. A Bean information class implements the BeanInfo interface. A BeanInfo class explicitly lists those Bean features that are to be exposed to the application builder tools. The Introspector recognises BeanInfo classes by their name. The name of a BeanInfo class is the name of the bean class followed by BeanInfo word e.g., for a bean named “Gizmo” the BeanInfo name would be “GizmoBeanInfo”.**

**•Properties:**

**Are a Bean’s appearance and behaviour characteristics that can be changed at design time.**

**•Customisation:**

**Beans expose properties so they can be customised during the design time.**

**A Bean developer can provide a *customizer* that helps another developer to customize a bean and configure his software. 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.**

**•Events:**

**Enables Beans to communicate and connect to each other.**

**•Persistence:**

**The capability of permanently stored property changes is known as Persistence. Beans can save and restore their state i.e., they need to be persistent. It enables developers to customise Beans in an app builder, and then retrieve those Beans, with customised features intact, for future use. JavaBeans uses Java Object Serialisation to support persistence. Serialisation is the process of writing the current state of an object to a stream. To serialise an object, the class must implement either java.io.Serialisable or java.io.Externalisable interface.**

**Beans that implement Serialisable are automatically saved and beans that implements Externalisable are responsible for saving themselves. The transient and static variables are not serialised i.e., these type of variables are not stored.**

**Beans can also be used just like any other Java class, manually (i.e., by hand programming), due to the basic Bean property, “Persistence”. Following are the two ways:**

**•Simply instantiate the Bean class just like any other class.**

**•If you have a customised Bean (through some graphic tool) saved into a serialised file (say mybean.ser file), then use the following to create an instance of the Customised Bean class...**

**try {**

**MyBean mybean = (MyBean)**

**Beans.instantiate(null, "mybean");**

**} catch (Exception e) {**

**}**

**•Connecting Events:**

**Beans, being primarily GUI components, generate and respond to events. The bean generating the event is referred to as event source and the bean listening for (and handling) the event is referred to as the event listener.**

**Bean Properties:**

**Bean properties can be categorised as follows...**

**1)Simple Property**

**are basic, independent, individual prperties like width, height, and colour.**

**2)Indexed Property**

**is a property that can take on an array of values.**

**3)Bound Property**

**is a property that alerts other objects when its value changes**

**4)Constrained Property**

**differs from Bound Property in that it notifies other objects of an impending change. Constrained properties give the notified objects the power to veto a property change.**

**Accessor Methods**

**1.Simple Property :**

**If, a bean has a property named**

**foo**

**of type**

**fooType**

**that can be read and written, it should have the following accessor methods:**

**public fooType getFoo( ) { return foo; }**

**public void setFoo(fooType fooValue) {**

**foo = fooValue; ...**

**}**

**If a property is boolean, getter methods are written using is instead of get**

**eg**

**isFoo( ).**

**2. Indexed Property :**

**public widgetType getWidget(int index)**

**public widgetType[] getWidget( )**

**public void setWidget(int index, widgetType widgetValue)**

**public void setWidget(widgetType[] widgetValues)**

**3.Bound Property :**

**Getter and setter methods for bound property are as described above based on whether it is simple or indexed. Bound properties**

**require certain objects to be notified when they change. The change notification isaccomplished through the generation of a**

**PropertyChangeEvent (defined in java.beans). Objects that want to be notified of a property change to a bound property must register as listeners. Accordingly, the bean that's implementing the bound property supplies methods of the form:**

**public void addPropertyChangeListener(ropertyChangeListener l)**

**public void removePropertyChangeListener(PropertyChangeListener l)**

**The preceding listener registeration methods do not identify specific bound properties. To register listeners for the PropertyChangeEvent of a specific property, the following methods must be provided:**

**public void addPropertyNameListener(PropertyChangeListener l)**

**public void emovePropertyNameListener(PropertyChangeListener l)**

**In the preceding methods, PropertyNameis replaced by the name of the bound property.**

**Objects that implement the PropertyChangeListener interface must implement the PropertyChange( ) method. This method is invoked by the bean for all registered listeners to inform them of a property change.**

**4.Constrained Property :**

**The previously discussed methods used with simple and indexed properties also apply to the constrained properties. In addition, the following event registeration methods provided:**

**public void addVetoableChangeListener(VetoableChangeListener l)**

**public void removeVetoableChangeListener(VetoableChangeListener l)**

**public void addPropertyNameListener(VetoableChangeListener l)**

**public void removePropertyNameListener(VetoableChangeListener l)**

**Objects that implement the VetoableChangeListener interface must implement the vetoableChange( ) method. This method is invoked by the bean for all of its registered listeners to inform them of a property change. Any object that does not approve of a**

**property change can throw a PropertyVetoException within its vetoableChange( ) method to inform the bean whose constrained property was changed that the change was not approved.**