How to be a Good Bean

- A JavaBeans component, or simply **a Bean**, is a reusable software component that can be manipulated visually in a builder tool.
- The JavaBeans 1.0 architecture specifies how a JavaBeans component exposes its features so tools can manipulate them to attain the needs of users.
- Commercial quality JavaBeans components and tools: Components that can be used and reused by different users in different tools, can interoperate with other components from other vendors, and are robust and functionally complete.

Creating JavaBeans Components

Two rules implied by the JavaBeans architecture include:

- the Bean class must provide zero-argument constructors so it can be created using **Beans.instantiate()**, and
- the Bean must support persistence, by implementing either Serializable or Externalizable.
- And, of course, the usual considerations need to be taken for persistence, including the use of the **transient** keyword when using the default read and write methods.

Bean Basics

- Any object that conforms to certain rules can be a bean.
  - No Bean-superclass
- Many beans are AWT components
  - Invisible beans may be useful
- A bean is characterized by properties, events, and methods it exports.
- A bean communicates by generating events.
  - Event model is based on java.util.EventObject and java.util.EventListener interfaces

Event model

- A bean defines an event if it provides add and remove methods for registering and deregistering listener objects for that event
- An application that wants to be notified when an event of that type occurs uses these methodes to register an event listener of the appropriate type
- When the event occurs, the bean notifies all registered listeners by passing an event object that describes the event to a methid defined by the event listener interface.
- Unicast vs. Multicast events

Beans - conventions

- Class name
  - No restrictions on the class name of a bean
- Superclass
  - A bean can extend any other class
  - Often AWT or Swing components
- Instantiation
  - Must provide no-parameter constructor or
  - File that contains serialized instance of a prototype bean (.ser-ext.)
- Bean name
  - Name of the class that implements it or name of the file that holds serialized instance of the bean („/“ converted to „.“ ; no .ser-ext.)

Methods

- Public methods describe the Bean's behavior.
  - Excluding methods that get/set property values
  - Excluding methods that register/remove event listeners
- Public methods are exposed by the builder tool.
- The methods are used by the builder tool or the user to construct connections between Beans.

Properties

A bean defines a property p of type T if it has a accessor method adhering to the following pattern:

- Getter:
  - public T getP()
- Boolean getter:
  - public boolean isP()
- Setter:
  - public void setP(T)
- Exceptions:
  - Property accessor methods can throw any type of checked or unchecked exceptions

Indexed Properties

- Property of array type
- Array getter:
  - public T[] getP()
- Element getter:
  - public T getP(int)
- Array setter:
  - public void setP(T[])
- Element setter
  - public void setP(int, T)
- Exceptions:
  - In particular ArrayIndexOutOfBoundsExceptions, ...

Bound Properties

- Accessor methods...
- Introspection
  - Implement BeanInfo class with PropertyDescriptor object for property
  - isBound() method should return true
- Listener registration
  - public void addPropertyChangeListener(PropertyChangeListener);
  - public void removePropertyChangeListener(PropertyChangeListsner);
- Notififcation
  - When value of bound property changes, bean should pass PropertyChangeEvent to propertyChange() method of every registered PropertyChangeListener

Constrained Properties

- When a property is about to change the listener is invoked through **VetoableChangeListener** and given an opportunity to veto the proposed change.
- Then when the change has actually happened, the (same) listener is invoked through **PropertyChangeListener** and the listener can react to the change.

Events

- You can use BeanInfo to facilitate use of the Bean’s events.
- The BeanInfo can also tailor access to the Bean's events by using the **hidden** and **expert** keywords.
  - Avoid misusing the event model--it is a notification scheme.
  - Use default signatures for the event methods. This accelerates tool interaction.

BeanInfo

There are several aspects to creating good BeanInfo:

- For each feature (method, event, or property) decide whether it should be exposed at all to Bean users, and if so whether should be marked as hidden or expert.
- Where applicable, provide a display name for the feature. Note that, for localization purposes, this name should be extracted from a resource bundle, so that different locales get useful localized feature names.
- Provide quality iconic representations. This makes the Bean easier to use and more recognizable within the builder tool.

Property Editors

- Property editors allow for better manipulation of the Bean's properties. It allows you to customize your JavaBeans component on the type level of the properties.
- A property editor can be a Bean in itself and can be packaged together with the Bean it belongs to.
  - You should deliver either a custom editor,
  - or paintable and
  - use the **getAsText()** method as last resort.
- A property editor can be linked to customizers via either BeanInfo or **PropertyEditorManager**.

Customizers

- A Bean may need to be represented in a specific manner to users, or may need to be configured in a certain sequence. Builder tools have no advance knowledge of this.
- By providing a Customizer the JavaBeans component developer can address these situations. The Customizer has full access to the Bean and is normally packaged with the Bean.
  - The Customizer can be a full-fledged JavaBeans component itself.
  - A Customizer can set a private state of the Bean that the property sheet and methods and events do not have access to.
- Customizers do not override property editors. By using BeanInfo you can associate property editors with customizers.

Packaging

- JavaBeans components are delivered by means of a JAR file.
- A JAR that contains a Bean must include
  - a Manifest declaring the Beans it contains,
  - be it a class or a persistent representation of a prototype of the Bean.
- In the absence of additional information, a Bean packaged in a given JAR file may require all the files in that JAR for its successful operation;
  - thus a builder tool that uses JavaBeans components from a series of JAR files will need to include all files in these JARs in the delivered products (be them Beans, applets, or applications).

Packaging (contd.)

- The JavaBeans 1.01 specification provides a mechanism for refining this through the **Design-Time-Only** and **Depends-On** tags.
  - Beans developers should consider using these headers for the benefit of tools. Refer to the 1.01 specification for details.
- The JAR can also contain any other files--such as images, sounds, HTML help files--that the Bean relies on.
- The JAR can also contain different localized versions of this information. To support deployment of the Bean in a builder tool the JavaBeans component developer can insert Javadoc information into the JAR as well.