301AA - Advanced Programming
[AP-2017]

Lecturer: Andrea Corradini andrea@di.unipi.it
Tutor: Lillo Galletta galletta@di.unipi.it

Department of Computer Science, Pisa
Academic Year 2017/18

AP-2017-09: Java based Components
Overview

• Kinds of components in Java
• JavaBeans: design and deployment
  – Properties
    • Property design pattern
  – Events
    • Connection-oriented programming
    • Observer design pattern
  – Serialization
  – Jar
  – Introspection (InfoBeans)
Components in Java SE (Standard Edition): Java Beans
Other Java Distributions

• Java EE (Enterprise Edition)
  – Suite of specifications for application servers
  – Around 20 implementations available
  – Reference implementation: Oracle Glassfish

• Java ME (Micro Edition)
  – embedded and mobile devices, e.g. micro-controllers, sensors, gateways, mobile phones, personal digital assistants (PDAs), TV set-top boxes, printers...
Components in Java EE (Enterprise Edition)

Client side
• JavaBeans
• Applets
• Application Components

Web server tier
• Servlets
• JSPs

Application tier:
• Stateless session EJB
• Stateful session EJB
• Entity EJB
• Message-driven EJB
Components in Java EE (Enterprise Edition)

Client side
- JavaBeans
- Applets
- Application Components

Web server tier
- Servlets
- JSPs

Application tier:
- Stateless session EJB
- Stateful session EJB
- Entity EJB
- Message-driven EJB

### Client tier
- Web browser + applets

### Web server tier
- JSP container + JSPs
  - Rich clients + application client components
  - Web service clients

### App server tier
- EJB container + Entity beans
  - Stateful & stateless session beans
  - Message-driven beans

### Backend tier
- Databases
- Legacy apps etc.

Naming and directories (JNDI)

Messaging (JMS)
The JavaBeans API (1996)

**Goal**: to define a software component model for Java, allowing vendors to create and ship Java components that can be composed together into applications by end users.

Design goals:

- **Granularity**: from small (eg. a button in a GUI) to medium (eg. a spreadsheet as part of a larger document)
  - Similar to OLE Control or ActiveX APIs
- **Portability**: Ok in Java based application servers. Bridges defined to other component models (like OpenDoc, OLE/COM/ActiveX)
- **Uniformity and Simplicity**: The API should be simple to be supported on different platforms. Strong support for small component, with reasonable defaults.
What are Java Beans?

“A Java Bean is a **reusable software component** that can be **manipulated visually** in a **builder tool**.”

- **Sample tools**: builders for web pages, visual applications, GUI layout, server applications. Also document editors.
- A bean typically has a GUI representation, but not necessarily
  - Invisible beans
- Any Java class can be recognized as a bean in a tool provided that
  - Has a public default constructor (no arguments)
  - Implements the interface `java.io.Serializable`
  - Is in a `jar` file with *manifest file* containing
    *Java-Bean: True*
JavaBeans as Software Components

- Beans are binary building blocks (class files)
- Development vs. deployment (customization)
- Beans can be assembled to build a new bean or a new application, applet, ...
- Writing glue code to wire beans together
- Client side bean vs. beans for business logic process in MVC on server
- Bean on server are not visible
Sample Reusable Components

Button Beans

Slider Bean

An application constructed from Beans
JavaBeans common features

- Support for “**properties**”, both for customization and for programmatic use
- Support for “**events**”: simple communication metaphor that can be used to connect several beans
- Support for “**customization**”: in the builder the user can customize the appearance and behaviour of the bean
- Support for **persistence**: a bean can be customized in an application builder and then have its customized state saved away and reloaded later
- Support for “**introspection**”: a builder tool can analyze how the bean works

Emphasis on GUI, but textual programming also possible using the existing API
Design time vs. run-time

• A bean must be able to run in the design environment of a builder tool providing means to the user to customize aspect and behaviour.

• At run-time there is less need for customization.

• Possible solution: design-time information for customization is separated form run-time information, and not loaded at run-time.

   – <BeanName>BeanInfo.java class
Simple Properties

• Discrete named attributes that can affect a bean instance’s appearance or behaviour
• Property X (and its type) determined by public setter (setX) and /or getter (getX) methods
• Can be changed at design time (customization) or run-time (application logic)
• Example property: background

```java
cpyublic java.awt.Color getBackground ();
cpyublic void setBackground (java.awt.Color color);
```
How can a builder identify the properties of a bean?
Introspection

• Process of analyzing a bean to determine the capability
• Allows application builder tool to present info about a component to software designer
• `<BeanName>BeanInfo` class to explicitly infer info on a bean
• Implicit method: based on reflection, naming conventions, and design patterns
Design Pattern for Simple Properties

From pair of methods:

```java
public <PropertyType> get<PropertyName>();
public void set<PropertyName>(<PropertyType> a);
```

infer existence of property `propertyName` of type `PropertyType`

• If only the getter (setter) method is present then the property is read-only (write-only)
Pattern for Indexed Properties

• If a property is an array, setter/getter methods can take an index or the whole array

```
public java.awt.Color getSpectrum (int index);
public java.awt.Color[] getSpectrum ();
public void setSpectrum (int index, java.awt.Color color);
public void setSpectrum (java.awt.Color[] colors);
```

• From these methods, by introspection the builder infers the existence of property spectrum of type java.awt.Color[]
Bound and Constrained Property

- A *bound property* generates an event when the property is changed.
- A *constrained property* can only change value if none of the registered observers poses a veto.

⇒ We present them after the event-based communication mechanism.
Connection-oriented programming

• Paradigm for gluing together components in a builder tool
• Based on the Observer design pattern
• Adequate for GUIs

**Diagram:**

```
Subject
  +Attach(in Observer)
  +Detach(in Observer)
  +Notify()  

ConcreteSubject
  -subjectState
  +GetState()  
  +setState()

ConcreteObserver
  -observerState
  +Update()  

Observers
  1  *

For all o in observers {
  o->Update()
}

Return subjectState

observerState = subject->GetState()
```

**Figure 9.1** The class diagram of the Observer pattern, with "notes" regarding implementations.
Events

• **Observer** design pattern also known as Publish-Subscribe

• In Java based on Events and Event Listeners

• An **event** is an object created by an **event source** and propagated to the registered **event listeners**

• Multicast semantics by default. Unicast semantics can be enforced by tagging the event source.
Design Pattern for Events

Based on methods for (un)registering listeners. From

```java
public void add<br>    <br><EventListType>(<EventListType> a)
```

```java
public void remove<br>    <br><EventListType>(<EventListType> a)
```

infer that the object is source of an event; the name is extracted from `EventListType`.

Example: from

```java
public void addUserSleepsListener (UserSleepsListener l);
```

```java
public void removeUserSleepsListener (UserSleepsListener l);
```

infers that the class generates a `UserSleeps` event
Unicast event sources

• Unicast semantics is assumed if the **add** method is declared to throw `java.util.TooManyListenersException`

• Example:

  ```java
  public void addJackListener(JackListener t) throws java.util.TooManyListenersException;
  public void removeJackListener(JackListener t);
  ```

defines a unicast event source for the “JackListener” interface.
Event Adaptors

- Placed between the event source and a listener
- Is at the same time listener and source
- Examples of uses of adaptors:
  - Implementing an event queuing mechanism between sources and listeners.
  - Acting as a filter.
  - Demultiplexing multiple event sources onto a single event listener.
  - Acting as a generic “wiring manager” between sources and listeners.
Event Adaptors

6.7 Event Adaptors

Event adaptors are an extremely important part of the Java event model. Particular applications or application builder tools may choose to use a standard set of event adaptors to interpose between event sources and event listeners to provide additional policy on event delivery.

6.7.1 Event Adaptor Overview

When additional behavior is required during event delivery, an intermediary "event adaptor" class may be defined, and interposed between an event source and the real event listener.

```
public synchronized void addFooListener(FooListener fel);
```

class XyzListener implements FooListener {
    void fooHappened(FooEvent fe) {
        cDestination.doIt(fe);
    }
}

Overview of Event Adaptor Model.
Demultiplexing multiple event sources

In the example (see the diagram and code below) a DialogBox object has two push buttons "OK" and "Cancel", both of which fire a `buttonPushed(PBEvent pbe)` method. The DialogBox is designed to invoke the methods, `doOKAction()` when the "OK" button fires, and `doCancelAction()` when the "Cancel" button fires.

The DialogBox defines two classes, `OKButtonAdaptor` and `CancelButtonAdaptor` that both implement the `PBListener` interface but dispatch the incoming notification to their respective action methods.

As a side effect of instantiation of the DialogBox, instances of the private adaptors are also created and registered with the `PushButton` instances, resulting in the appropriate event flow and mapping.

```java
// Adaptor to map "Cancel Pressed" events onto doCancelAction()
class CancelAdaptor implements PushButtonExampleListener {
    private Dialog dialog;
    public CancelAdaptor(Dialog dest) {
        dialog = dest;
    }
    public void buttonPushed(PushButtonExampleEvent pbe) {
        dialog.doCancelAction();
    }
}
```

```java
// OK Button
buttonPushed(PBEvent pbe) {
    dialog.doOKAction();
}
```

```java
// Cancel Button
buttonPushed(PBEvent pbe) {
    dialog.doCancelAction();
}
```
Back to Bound Properties

• Can generate 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
• Bean with bound property: Event source
• Bean implementing listener: event target
• Helper classes in the API to simplify implementation
Implement Bound Property in a Bean

1. Import `java.beans` package
2. Instantiate a `PropertyChangeSupport` helper object
   ```
   private PropertyChangeSupport changes = new PropertyChangeSupport(this);
   ```
3. Implement methods to maintain the property change listener list:
   ```
   public void addPropertyChangeListener(PropertyChangeListener l)
   { changes.addPropertyChangeListener(l); }
   ```
   (also `removePropertyChangeListener` method is needed)
Implement Bound Property in a Bean

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

```java
public void setX(int newX){
    int oldx = x;
    x = newX;
    changes.firePropertyChange("x", oldX, newX);
}
```
Implement Bound Property Listener

1. Listener bean must implement the `PropertyChangeListener` interface

   ```java
   public class MyClass implements PropertyChangeListener, Serializable
   ```

2. It must override this method:

   ```java
   public abstract void propertyChange(PropertyChangeEvent evt)
   ```

3. Registration

   ```java
   Button button = new OurButton();
   button.addPropertyChangeListener(aButtonListener);
   ...
Constrained Property

- It generates an event when an attempt is made to change its value
- The event type is `PropertyChangeEvent`
- The event is sent to objects that previously registered an interest in receiving such notification
- Those other objects have the ability to veto the proposed change
- This allows a bean to operate differently according to the runtime environment
Three Parts in Implementation of Constrained Property

1. Source bean containing one or more constrained properties

2. Listener objects that implement the `VetoableChangeListener` interface. This object either accepts or rejects the proposed change.

3. `PropertyChangeEvent` object containing property name, old value, new value.
Implement Constrained Property in a Bean

Bean with constrained property must

1. Allow VetoableChangeListener object to register and unregister its interest in receiving notifications
2. Fire property change at those registered listeners. The event is fired before the actual property change takes place
Implement Constrained Property in a Bean

1. Import java.beans package
2. Instantiate a VetoableChangeSupport object:
   ```java
   private VetoableChangeSupport vetos = new VetoableChangeSupport(this);
   ```
3. Implement methods to maintain the property change listener list:
   ```java
   public void addVetoableChangeListener(VetoableChangeListener l) {
       vetos.addVetoableChangeListener(l);
   }
   ```
4. Write a property’s setter method to fire a property change event:

```java
public void setX(int newX) {
    int oldX = X;
    vetos.fireVetoableChange("X", oldX, newX);
    //if no veto there
    X = newX;
    changes.firePropertyChange("X", oldX, newX);
}
```
Implementing Constrained Property Listeners

1. Implements the `VetoableChangeListener` interface which has an abstract method
   
   ```java
   void vetoChange(PropertyChangeEvent evt)
   ```

2. Override this abstract method. This is the method that will be called by the source bean on each object in the listener list kept by `VetoableChangeListenerSupport` object
Builder Tools and Properties

• Discover Bean’s Properties
• Determine properties’ read/write attributes
• Determine property types
• Locate property editors
• Display property sheet
• Alter properties
Persistence through Serialization

• Beans use Java’s object Serialization API to provide a great medium-weight solution for persistence.

• The `Beans.instantiate` method is normally used by builder tools to recreate a Bean from a serialized Bean source
  – i.e., cut and paste uses serialization to copy a Bean’s data
  – or Beans can connect or communicate with each other in an application or across the web
Summary

- JavaBean is a platform-neutral component architecture for reusable software component.
- It is a black box component to be used to build large component or application.
- Property, method, event, introspector, customizer are parts of javabeaean API.