

CMSC433, Spring 2001 JavaBeans, with examples

Alan Sussman
May 8, 2001

Administrivia

- Project 6 spec has been updated
 - file sharing registration now takes a URL object, not a File
 - don't need to run a web server, can put shared files on WAM web site, or wherever
 - provided listener implementation just prints URL
 - sample startup script posted

CMCS 433, Spring 2001 - Alan Sussman

2

Last time

- JavaBeans
 - Software components in Java
 - Event model
 - same as for AWT and Swing GUI libraries
 - listeners register with a bean to be notified of events
 - Properties – *get* and *set*
 - *bound* – other beans notified of a property change
 - *constrained* – other beans can veto a property change
 - Introspection
 - use Java reflection to find out about a bean's properties, events, other methods
 - Persistence
 - all beans can be serialized and deserialized

CMCS 433, Spring 2001 - Alan Sussman

3

Simple Bean example

```
import java.awt.*;
import java.io.Serializable;
public class SimpleBean extends Canvas
    implements Serializable {
    //Constructor sets inherited properties
    public SimpleBean(){
        setSize(60,40);
        setBackground(Color.red);
    }
}
```

CMCS 433, Spring 2001 - Alan Sussman

4

Properties

- If a component supports functions:
 - `public void setMyValue(int v)`
 - `public int getMyValue()`
- It has a MyValue property of type int
- For boolean types, getter function can be named `is<Prop>()`
- Can have read-only, read/write or write-only properties
 - don't have to define *both* getter and setter method

CMCS 433, Spring 2001 - Alan Sussman

5

Example, with Simple Property

```
import java.awt.*; import java.io.Serializable;
public class SimpleBean extends Canvas
    implements Serializable{
    private Color color = Color.green;
    // property getter method
    public Color getColor(){
        return color;
    }
    // property setter method. Sets new SimpleBean
    // color and repaints.
    public void setColor(Color newColor){
        color = newColor;
        repaint();
    }
}
```

CMCS 433, Spring 2001 - Alan Sussman

6

Simple Property, cont.

```
public void paint(Graphics g) {
    g.setColor(color);
    g.fillRect(20, 5, 20, 30);
}
//Constructor sets inherited properties
public SimpleBean(){
    setSize(60,40);
    setBackground(Color.red);
}
}
```

CMCS 433, Spring 2001 - Alan Sussman

7

Java Bean Event Patterns

- A Bean Event must extend
 - class java.util.EventObject {
public EventObject(Object src);
public Object getSource();
}
- Name should end in Event
 - e.g., tempChangeEvent

CMCS 433, Spring 2001 - Alan Sussman

8

Event Listeners

- must implement java.util.EventListener
 - just a marker interface
- have event-Listener methods
 - void <eventName>(<EventObjectType> e);
- interface TempChangeListener {
void tempChanged(TempChangedEvent e);
}

CMCS 433, Spring 2001 - Alan Sussman

9

Event sources

- Event sources fire events
- Have methods to attach/detach Listeners
 - public void add<ListenerType>(ListenerType ls);
 - public void remove<ListenerType>(ListenerType ls);

CMCS 433, Spring 2001 - Alan Sussman

10

Event Adapters

- Easy to construct event adapters
 - For example, an adapter that receives temperatureChanged events, and generates temperatureIncreased and temperatureDecreasedEvents

CMCS 433, Spring 2001 - Alan Sussman

11

Bound properties

- Can set things up so that changes to bean property are indicated by an event
 - after the change occurs
 - events are a subtype of java.beans.PropertyChangeEvent
 - Listeners implement PropertyChangeListener and the propertyChange method is invoked when the event is fired
 - One Listener for all change events on the bean
 - may optionally support listeners for specific properties

CMCS 433, Spring 2001 - Alan Sussman

12

Bound Property support

- Convenience class **PropertyChangeSupport**
 - implements methods to add/remove `PropertyChangeListener`s, and fire `PropertyChangeEvent` objects at listeners when the bound property changes

CMCS 433, Spring 2001 - Alan Sussman

13

Implementing a Bound Property

```
import java.beans.*
public class Bound ... {
    // instantiate PropertyChangeSupport object
    private PropertyChangeSupport changes =
        new PropertyChangeSupport(this);
    // methods to implement property change listener list
    public void addPropertyChangeListener(
        PropertyChangeListener l) {
        changes.addPropertyChangeListener(l); }
    public void removePropertyChangeListener(
        PropertyChangeListener l) {
        changes.removePropertyChangeListener(l); }
```

CMCS 433, Spring 2001 - Alan Sussman

14

Bound Properties, cont.

```
// modify property setter method to fire PropertyChangeEvent
public void setLabel(String newLabel) {
    String oldLabel = label;
    label = newLabel;
    sizeToFit();
    changes.firePropertyChange("label", oldLabel, newLabel); }
```

```
// this builds a PropertyChangeEvent object, and calls
// propertyChange(PropertyChangeEvent pce) on each registered
// listener
public void firePropertyChange(String propertyName,
    Object oldValue, Object newValue)
```

CMCS 433, Spring 2001 - Alan Sussman

15

Creating a Listener

```
// implement the PropertyChangeListener interface
public class MyClass
    implements java.beans.PropertyChangeListener,
        java.io.Serializable {
    void propertyChange(PropertyChangeEvent evt) {
        // handle a property change event
        // e.g., call a setter method in the listener class
    }
}
```

```
// and register the listener with the source Bean
button.addPropertyChangeListener(aButtonListener);
```

CMCS 433, Spring 2001 - Alan Sussman

16

Constrained Properties

- Source Bean contains one or more *constrained* properties
 - should also usually be bound properties
- Listeners can veto property changes
 - before the actual property change occurs
 - implement `VetoableChangeListener` interface
 - Listener throws `PropertyVetoException`
 - `set<Property>` method throws ...

CMCS 433, Spring 2001 - Alan Sussman

17

Constrained Properties

```
import java.beans.*
public class Constrained ... {
    // instantiate VetoableChangeSupport object
    private VetoableChangeSupport vetos =
        new VetoableChangeSupport(this);
    // methods to implement property change listener list
    public void addVetoableChangeListener(
        VetoableChangeListener l) {
        vetos.addVetoableChangeListener(l); }
    public void removeVetoableChangeListener(
        VetoableChangeListener l) {
        vetos.removeVetoableChangeListener(l); }
```

CMCS 433, Spring 2001 - Alan Sussman

18

Constrained Properties, cont.

```
// modify property setter method to fire PropertyChangeEvent
// including adding throws clause
public void setPriceInCents(int newPriceInCents)
    throws PropertyVetoException {
    int oldPriceInCents = ourPriceInCents;
    // First tell the vetoers about the change.
    // If anyone objects, we don't catch the exception
    // but just let it pass on to our caller.
    vetos.fireVetoableChange("priceInCents",
        new Integer(oldPriceInCents),
        new Integer(newPriceInCents));
    // No-one vetoed, so go ahead and make the change.
    ourPriceInCents = newPriceInCents;
    changes.firePropertyChange("priceInCents",
        new Integer(oldPriceInCents),
        new Integer(newPriceInCents));
}
```

CMCS 433, Spring 2001 - Alan Sussman

19

Constrained Properties, cont.

```
// this builds a PropertyChangeEvent object, and calls
// vetoableChange(PropertyChangeEvent pce) on each registered
// listener
public void fireVetoableChange(String propertyName,
    Object oldValue,
    Object newValue)
    throws PropertyVetoException
```

CMCS 433, Spring 2001 - Alan Sussman

20

Creating a Listener

- Same as for PropertyChangeListener
 - listener Bean implements **VetoableChangeListener** interface
 - with method **vetoableChange(PropertyEvent) throws PropertyVetoException;**
 - called by source Bean on each registered listener object, and exercises veto power by throwing the **PropertyVetoException**

CMCS 433, Spring 2001 - Alan Sussman

21

Serialization and Persistence

- Can manipulate Java Beans in a builder tool
- Doesn't help if can't distribute the beans
- *Serialize* the beans
 - Bean must implement **java.io.Serializable** or **java.io.Externalizable** (to get complete control over the serialization)
- Application loads beans from Serialized form

CMCS 433, Spring 2001 - Alan Sussman

22

Default Serialization

- Beans that implement **Serializable** must have a *no-argument constructor*
 - to call when rebuilding the object
- Don't need to implement **Serializable** if already implemented in a superclass
 - unless need to change the way it works
- All fields except *static* and *transient* ones are serialized
 - default serialization ignores those fields
 - *transient* also can mark an entire class as not serializable (!?!)

CMCS 433, Spring 2001 - Alan Sussman

23