Java Beans

Acknowledgements: Most slides courtesy of Jens Gustavsson, Mikhail Chalabine, Peter Bunus

Outline

- What a Java Bean component is
- Similarities and differences between beans and regular objects
- How to develop a Java Beans component that follows the naming pattern
- The Java event delegation model
- Another encounter with reflection and introspection
- Creating custom event classes and listener interfaces
What is a JavaBean?

**JavaBeans – a client-side component model for Java**

- Written in Java
  - Portable
  - Platform-independent
- Limited to a single Java process (no distribution transparency)
- Mostly used for GUI programming
- API introduced in February 1997

**Java Bean = reusable Java component**

**JavaBeans!≠ Enterprise JavaBeans**

- The next lecture will be dedicated to EJB
A reusable component in Java

Class
- Hides implementation, conform to interfaces, encapsulates data
- Is written to a standard (component specification)
  - Implements serializable interface (persistence)
  - No argument constructor (instantiation via reflection)
  - Design Patterns or BeanInfo class (introspection)
  - Core Features (method Properties, events)

A JavaBeans component is a serializable public class with a public no-arg constructor
A bean has methods, properties and events.

- **Properties** represent the part of a bean's internal state that is customizable from the outside
- A bean fires **events** at interesting points during its execution
- A bean exposes its methods, properties, and events.

You can have all this in a normal Java class as well – but JavaBeans defines a standard such that these things can be explored, e.g., by a GUI builder.
JavaBean and Component Types

Software components are self-contained, reusable software units

**Visual software components**

- Using visual application builder tools, visual software components can be composed into applets, applications, servlets, and composite components.
- You perform this composition within a graphical user interface, and you can immediately see the results of your work.

**Non-visual software components**

- Capture business logic or state
Example JavaBean Component Types

Visual Components
- Used in SWING, AWT
- Visual application builders (visual composition)
  - Work flow: load, customize (size, color), save (persist)
  - Eclipse WindowBuilder GUI Builder
  - NetBeans

Non-Visual Components
- In Java2EE
- Capture Business logic or state

Examples of Beans
- GUI (graphical user interface component)
- Non-visual beans such as a spell checker
- Animation applet
- Spreadsheet application
JavaBean Core Features

**Properties**— the appearance and behavior characteristic of a bean that can be changed at design time

**Events**— Beans use events to communicate with other beans

**Methods**— methods are similar to the Java methods
JavaBean Properties
**Bean’s features: Properties**

- A **bean property** is a named attribute of a bean that can affect its behavior or appearance.
- Examples include *color, label, font, font size*, and *display size*.

```
import java.io.Serializable

public class MyJavaBean implements Serializable
{
    private String first_name;
    private float income;
    ...
```

**Visual Components**

- **Builder tools** can discover and expose
- **Customization** – modifying appearance or behavior at design time by
  - Property editors (visual, programmable)
  - Bean customizers (visual, programmable)
Bean’s features: Properties

Semantically, a bean consists of a collection of properties (plus some other methods)

Specification suggests “getters” and “setters”

```java
public String getFirst_name(){
    return first_name;
}

public String setFirst_name(String first_name ){
    this.first_name = first_name;
}

public String getIncome(){
    return income;
}

public void setIncome(float income ){
    this.income = income;
}
```
Bean’s features: Properties

Properties should not be confused with instance variables

Even though instance variables are often mapped directly to property names, properties of a bean are not required to correspond directly to instance variables.

```java
public class DiceBean {
    private java.util.Random rand;

    public DiceBean() {
        rand = new Random();
    }

    public int getDiceRoll() { // return a number between 1 and 6
        return rand.nextInt(6) + 1;
    }

    public void setDiceRoll(int i) {
        // do nothing
    }
}
```
**Types of Properties**

- **Simple**—A bean property with a single value whose changes are independent of changes in any other property.

- **Indexed**—A bean property that supports a range of values instead of a single value.

- **Bound**—A bean property for which a change to the property results in a notification being sent to some other bean.

- **Constrained**—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.
import java.util.*;

public class StatBean {
  private double[] numbers;
  public StatBean() {numbers = new double[0];}

  public double getAverage() { .. }
  public double getSum() { .. }

  public double[] getNumbers() {
    return numbers;
  }

  public double getNumbers(int index) {
    return numbers[index];
  }

  public void setNumbers(double[] numbers) {
    this.numbers = numbers;
  }

  public void setNumbers(int index, double value) {
    numbers[index] = value;
  }

  public int getNumbersSize() {
    return numbers.length;
  }
}
Bean Boolean Properties

- Properties that are either true or false
- Setter/getter methods conventions

```java
public boolean isProperty();
public void setProperty( boolean b );

public boolean isEnabled();
public void setEnabled( boolean b );

public boolean isAuthorized();
public void setAuthorized( boolean b );
```
JavaBean Events
A bean may communicate with other beans. The Java event delegation model provides the foundation for beans to send, receive, and handle events.

When something happens to a bean, such as a mouse click on a javax.swing.JButton bean, an event object is created to encapsulate information pertaining to the event. The bean passes the event object to the interested beans for the event to be processed.

Events are typically generated by Java GUI components such as javax.swing.JButton, but are not limited to GUI components.
Bean features: Event model

- **Fire**(send) / **handle**(receive)
  - Component broadcast events and the underlying framework delivers the event to the components to be notified

- **Sources**
  - Defines and fire events
  - Define methods for registering listeners

- **Listeners**
  - Get notified of events
  - Register using methods defined by sources

---

```
java.awt.event.MouseEvent
```

---

---

```
--Click--
```

---

source

---

listener
Do you Remember The Observer Pattern?

Here is the Subject interface. Objects use this interface to register as observers and also to remove themselves as observers.

Each subject can have many observers.

All potential observers need to implement the Observer interface. This interface has just one method `Update()` that gets called when the Subject’s state changes.

Concrete observers can be any class that implements the Observer Interface. Each observer register with a concrete subject to receive updates.

Concrete subject may also have methods for setting and getting its state.

A concrete subject always implements the Subject interface. In addition to the register and remove methods, the concrete Subject implement a Notify() method that is used to update all the current observers whenever state changes.
Steps in Writing Event Handling

**Write Event class**
- Create your own custom event class, named `XXXEvent` or use an existing event class
- There are existing event classes (i.e. `ActionEvent`)

**Write Event listener** (Event handler or Event receiver)
- Write `XXXListener` interface and provide implementation class of it
- There are built-in listener interfaces (i.e. `ActionListener`)

**Write Event source** (Event generator)
- Add an `addXXXListener` and `removeXXXListener` methods, where `XXX` stands for the name of the event
- These methods are used by event listeners for registration
- There are built-in event source classes

**Write a listener**
- Register event listener to the event source through `addXXXListener()` method of the event source
Discovering Bean Features
Bean Features: Naming Convention

The specification of the JavaBean component model by Sun defines a set of conventions:

- Naming of methods, event, properties
- Confusingly, also called “Design Patterns” (unfortunate misuse of the term “design patterns”, should have been called “naming patterns”)

The bean programmer follows these standards in naming elements of the bean.

Visual tools use pre-defined Introspector classes to dynamically extract the BeanInfo metadata based on these conventions.
Things That Can be Found through Introspection

**Simple property**

```java
public void setPropertyName( PropertyType value );
public PropertyType PropertyName();
```

**Boolean property**

```java
public void setPropertyName( boolean value );
public boolean isPropertyName();
```

**Indexed property**

```java
public void setPropertyName( int index, PropertyType value );
public PropertyType getPropertyName( int index );
public void setPropertyName( PropertyType[] value );
public PropertyType[] getPropertyName();
```
Things That Can be Found through Introspection

**Multicast events**

- `public void addEventListenerType( EventListenerType l );`
- `public void removeEventListenerType( EventListenerType l );`

**Unicast events**

- `public void addEventListenerType( EventListenerType l ) throws TooManyListenersException;`
- `public void removeEventListenerType( EventListenerType l );`

**Methods**

- `public methods`
Bean Persistence
Bean Persistence

Through object serialization

**Object serialization** = converting an object into a byte stream.
- Recursive traversal of object structure (cf. depth-first search)
- Type checking is done at deserialization (type safety!)
- Serializable interface → need to provide serialize, deserialize methods
- Remark: Java JDK serialization is terribly slow!

**Persistence** = keeping it in nonvolatile storage (e.g. file, database).

Any applet, application, or tool that uses that bean can "reconstitute" a serialized object by **deserialization**.
- The object is then restored to its original state.

Serialized objects can travel across system boundaries.
- For example, a Java application can serialize a Frame window on a Microsoft Windows machine, the serialized file can be sent with e-mail to a Solaris machine, and then a Java application can restore the Frame window to the exact state which existed on the Microsoft Windows machine.
- Platform independent, but Java language versions must match.
Bean Persistence: XMLEncoder/XMLDecoder

Enable beans to be saved in XML format

The XMLEncoder class is assigned to write output files for textual representation of Serializable objects

```java
encoder = new XMLEncoder(new BufferedOutputStream(new FileOutputStream("Beanarchive.xml")));
encoder.writeObject(object);
```

XMLDecoder class reads an XML document that was created with XMLEncoder:

```java
decoder = newXMLDecoder(newBufferedInputStream(newFileInputStream("Beanarchive.xml" )));
Object object = decoder.readObject();
```
import java.awt.Color;
import java.beans.XMLDecoder;
import javax.swing.JLabel;
import java.io.Serializable;

public class SimpleBean extends JLabel implements Serializable {

    public SimpleBean() {
        setText( "Hello world!" );
        setOpaque( true );
        setBackground( Color.RED );
        setForeground( Color.YELLOW );
        setVerticalAlignment( CENTER );
        setHorizontalAlignment( CENTER );
    }
}

<xml version="1.0" encoding="UTF-8" ?>
<java>
  <object class="javax.swing.JFrame">
    <void method="add">
      <object class="java.awt.BorderLayout" field="CENTER"/>
      <object class="SimpleBean"/>
    </void>
    <void property="defaultCloseOperation">
      <object class="javax.swing.WindowConstants" field="DISPOSE_ON_CLOSE"/>
    </void>
    <void method="pack"/>
    <void property="visible">
      <boolean>true</boolean>
    </void>
  </object>
</java>