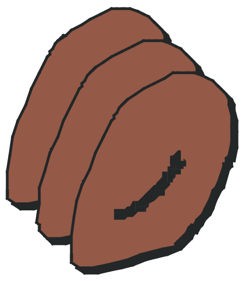
Writing JavaBeans

- What is a JavaBean?
- Properties, Methods and Events

What is a JavaBean?

- A JavaBean is a Java class that is designed to manipulated by tools or by other programs (.e. g. JavaServer Pages) in a standard fashion
- The JavaBean specification defines the JavaBean as the standard component architecture for Java

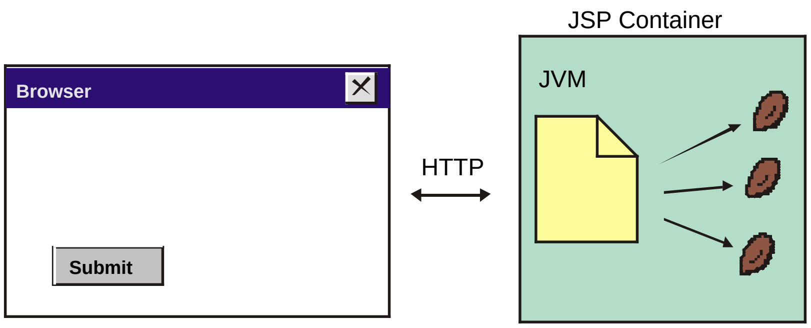


The JavaBean specification is available from:

http://java.sun.com/products/javabeans/docs/spec.html

Why Use JavaBeans?

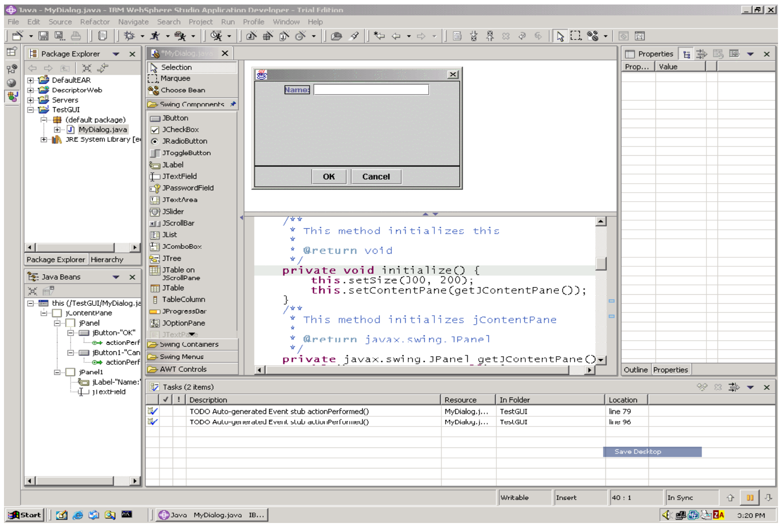
- Some development tools use JavaBeans to let you drag and drop to create graphical user interfaces (GUI)
- JavaServer Pages and other J2EE technologies can access JavaBeans that provide state and behavior



The JavaBean specification was initially created to facilitate GUI builders for rich clients, but since then, JavaBeans have more widely been used in server-side applications.

JavaBean GUI Toolkits

- IBM WebSphere Studio Application Developer
- Sun NetBeans
- Eclipse with Visual Editor Project Plugin
- Borland JBuilder
- JetBrains IntelliJ IDEA


Java - MyDialog.java - IBM WebSphere Studio Application Developer - Trial Edition
File Edit Source Refactor Navigate Search Project Run Profile Window Help
Package Explorer
Java Beans
Name:
OK
Cancel
Properties
Tasks (2 items)

```
/**
 * This method initializes this
 *
 * @return void
 */
private void initialize() {
    this.setSize(300, 200);
    this.setContentPane(getJContentPane());
}
/**
 * This method initializes jContentPane
 *
 * @return javax.swing.JPanel
 */
private javax.swing.JPanel getJContentPane()
```

Description	Resource	In Folder	Location
TODO Auto-generated Event stub actionPerformed()	MyDialog.j...	TestGUI	line 79
TODO Auto-generated Event stub actionPerformed()	MyDialog.j...	TestGUI	line 96

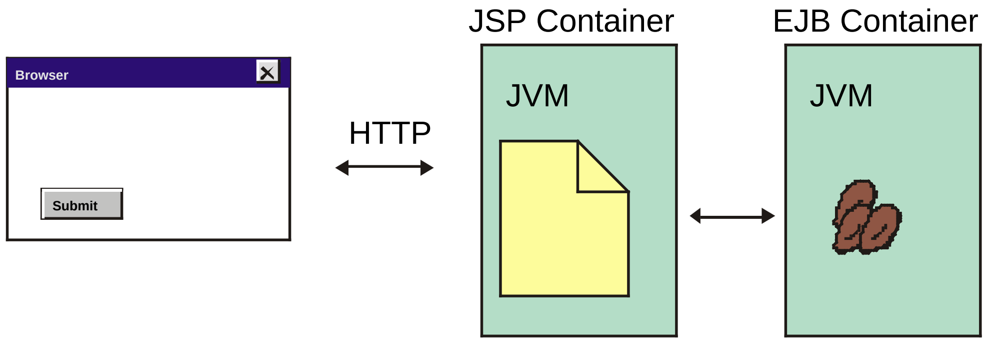
Sun also provides the free Bean Builder tool that is a demonstration program on how to use JavaBeans to build GUIs.

The Bean Builder is available from:

https://bean-builder.dev.java.net/

JavaBeans versus Enterprise JavaBeans

- Enterprise JavaBeans implement specific interfaces so they can run within an **EJB container** that provides services such as transactions, security and persistence
- EJBs can be invoked remotely, while "normal" JavaBeans run in the same JVM as the caller



The EJB specification is an integral part of J2EE and allows for components that execute remotely. We will not cover EJBs in this class, but you should be aware that it's very possible to design web applications that include JSPs, servlets and EJBs. Such designs are complex but scalable, since they let you separate the business logic onto remote servers. In addition, since the EJB container provides powerful services, applications that use EJBs can have truly enterprise-class robustness.

JavaBean Requirements

- A JavaBean is a Java class that follows a few simple conventions
- To be a proper JavaBean, a Java class should:
  - Implement the Serializable interface
  - Be in a named package
  - Expose properties, methods and events as necessary
  - Define a no-argument constructor
  - Optionally packaged into a JAR file

This page lists some of the conventions you should follow to create compliant JavaBeans. We will see a complete example in a few pages.

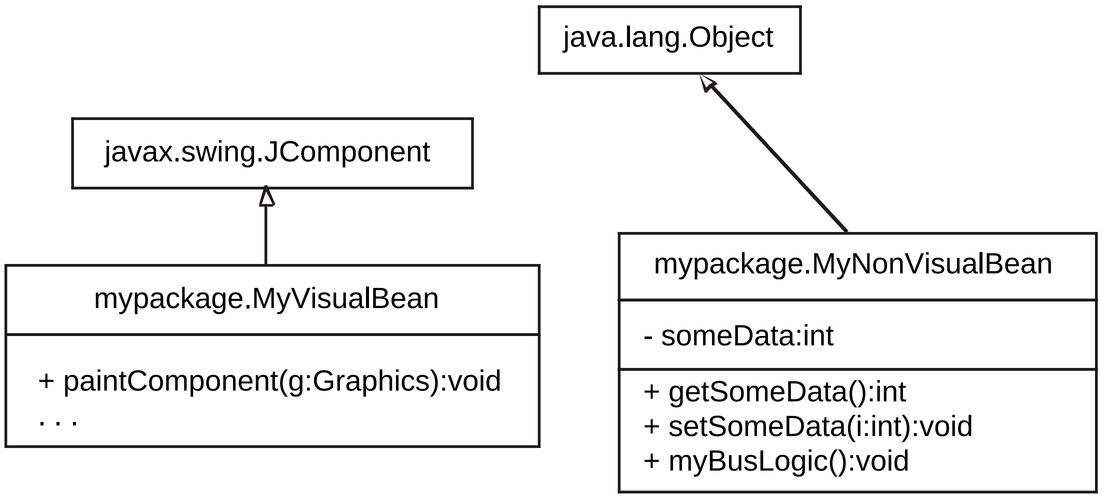
JavaBean Features

- A JavaBean can expose three kinds of features:
  - **Properties** , which are defined using get/set methods
  - **Methods**, which expose behavior
  - **Events**, which let the Bean call back to the outside world

The JavaBean specification provides precise definitions of how to code a JavaBean that provides these three kinds of features.

Visual versus Non-Visual Beans

- Visual beans are designed to appear in a graphical user interface (GUI)
- Non-visual beans generally define business logic and data



The Java class library defines many visual beans (e.g. JButton, JTextField and so forth) that you can use "out-of-the-box" and you can also create your own custom visual beans that integrate into GUI builders.

Non-visual beans generally contain business methods, data (properties) and perhaps fire events. Most GUI builders supply a mechanism so that GUIs can access non-visual beans.

What are Reflection and Introspection?

- The Java API includes a technique known as **reflection** that lets a program enumerate the public features of any Java class
- If the Java class follows some simple naming conventions, the examining program can make assumptions about the class using a process referred to as **introspection**
- For example, if a JavaBean has a two public methods, one named *getAddress* and one named *setAddress*, we can infer that the Bean defines a property named address

For example, a JSP container uses reflection and introspection to determine a Bean's features and thus check that the JSP author accesses the features correctly.

Writing JavaBean Methods

- A JavaBean method is simply a public method that exposes a behavior

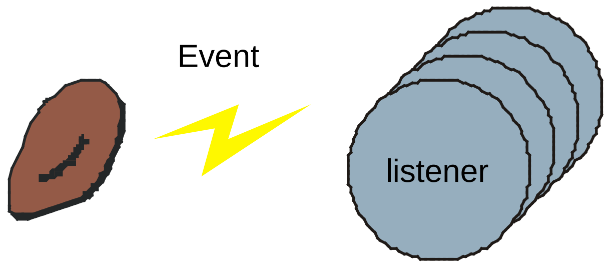
```
public boolean calcHonorRoll()
{
    if ( getGpa() > 3.5 )
        return true;
    else
        return false;
}
```

2 - 10

This JavaBean method is taken from a StudentBean class.

Introduction to Events

- Events allow JavaBeans to notify listeners of significant occurrences
- The following artifacts comprise an event:
    - An event-object class
    - An event listener interface
    - Register/deregister methods on the bean
    - Firing the event



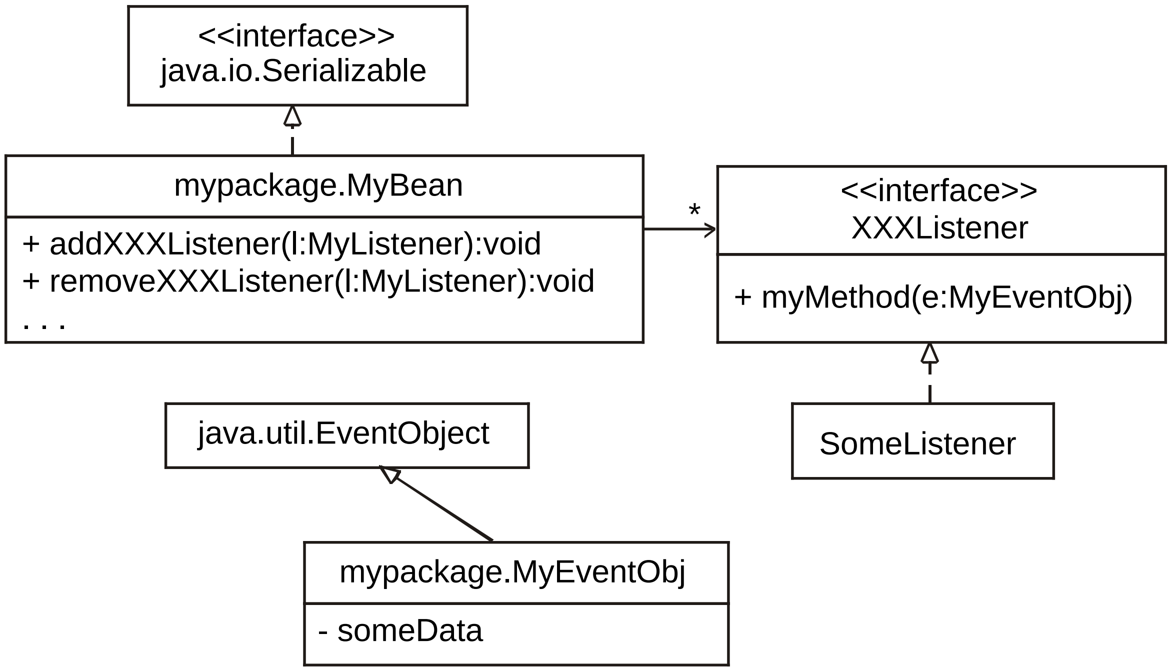
Events are almost the opposite of methods -- methods let the outside world invoke behavior on the bean, while an event is the bean invoking behavior on "listeners", which are objects that have registered an interest in the event.

Events are especially useful in GUIs, where a user-interface component, say a button, notifies another object when the user clicks the button.

Note that there can be multiple listeners on a given event. The bean notifies all registered listeners when the event occurs.

JavaBean events are a variation on the Observer Design Pattern from the Design Patterns book by Gamma, et. al.

Event Architecture



2 - 12

The idea is that the bean maintains a list of interested listeners, each of which is typed to the interface that we define. The bean provides two methods: one to add a listener to the list, another to remove. Note that the names of these two methods are very significant -- we use introspection to infer the event name from the method names. In other words, whatever you put as the 'XXX' placeholder above will be interpreted as the event name.

We also can define a class that describes the event -- note that we pass an instance of the event class in the interface methods. This is optional -- if you don't have any extra information to pass, you can use the java.util.EventObject class directly.

The Event Object Class

- Event objects carry information about the event

```
public class FlunkEvent extends java.util.EventObject
{
    private String courseName;

    public FlunkEvent(Object src, String coursName)
    {
        super(src);
        this.courseName = coursName;
    }
    public String getCourseName()
    {
        return courseName;
    }
    public void setCourseName(String courseName)
    {
        this.courseName = courseName;
    }
}
```

The Java class library (especially in the java.awt.event package) defines several event-object classes and you can write your own as we did here.

The standard EventObject superclass basically wraps a reference to the JavaBean that fired the event.

Event Listener Interfaces

- An event is essentially a callback from the JavaBean to the interested party
- The interested party must therefore define method(s) that the JavaBean can call
- We enforce this requirement using listener interfaces

```
public interface FlunkListener
  extends java.util.EventListener
{
  public void studentFlunked(FlunkEvent evt);
}
```

The listener interface defines the behavior(s) that any listener must provide.

Generally, the interface methods will take an event object as described on the last page.

Registering for an Event

- The JavaBean provides methods to allow interested listeners to add or remove themselves from the JavaBean's list of listeners
- It's the existence of the add/remove methods that allow tools to infer the event

These methods imply...	an event named:

addFlunkedOutListener()
removeFlunkedOutListener()
	FlunkedOut

Tools that examine JavaBeans use reflection and introspection to make inferences based on naming patterns. According to the JavaBean specification, if a bean provides methods named addXXXListener and removeXXXListener, we can infer an event named XXX.

Registering for an Event, cont'd

```
public class Student implements Serializable
{
. . .
    private Vector listeners = new Vector();

    public synchronized void
      addFlunkedOutListener (FlunkListener fl)
    {
        listeners.add(fl);
    }
    public synchronized void
      removeFlunkedOutListener(FlunkListener fl)
    {
        listeners.remove(fl);
    }
. . .
}
```

The code to implement registering and de-registering listeners is trivial. Note, however the type of the arguments -- it's the type of the listener interface!

Firing an Event

- When the significant occurrence happens, the JavaBean **fires** the event by traversing the listener list, calling the interface method(s) on each listener

```
protected void fireFlunkedOutEvent(String course)
{
    Vector copy = null;
    FlunkEvent evt = new FlunkEvent(this,course);

    synchronized(this)
    {
        copy = (Vector)listeners.clone();
    }
    Iterator iter = copy.iterator();
    while (iter.hasNext())
    {
     ((FlunkListener)iter.next()).studentFlunked(evt);
    }
}
```

The code shown here first copies the listener list in an atomic block (so that we don't get messed up if a listener attempts to add or remove itself while we're firing the event). Then it's a simple matter of traversing the list, calling the event method on each listener.

Of course, we need to call this method at some point -- that's sometimes the tricky part -- figuring out when to actually fire the event. We'll look at that in more detail for the Student bean in a moment.

Event Consumers

- Objects that are interested in the event must implement the event-listener interface and then register for the event

```
public class RespondToFlunk implements FlunkListener
{
    public void studentFlunked(FlunkEvent evt)
    {
        System.out.println("Flunking: " +
                evt.getCourseName());
    }

    public static void main(String[] args)
    {
        RespondToFlunk me = new RespondToFlunk();
        Student s = new Student();
        s.addFlunkedOutListener(me);
        s.setGpa(0.4);
    }
}
```

Here we show a simple class that demonstrates listening to an event. The class implements the listener interface and then registers an object. Then, when we set the GPA to a low value, the JavaBean will fire the FlunkedOut event.

JavaBean Properties

- A **property** exposes state from a JavaBean
- There are four kinds of properties defined in the JavaBean specification:
    - Simple Properties
    - Indexed Properties
    - Bound Properties
    - Constrained Properties

Properties represent the data of a JavaBean and are probably the most-used feature of JavaBeans.

Simple Properties

- A **property** is defined by the existence of "getter" and "setter" methods
- Note that the property name is case sensitive!
- A **readonly property** defines only a "getter" method

```
public String getAddress() {...}
public void setAddress ( String s ) {...}
public int getSerialNumber () {...}
```

Property Name	Property Type
address	String
serialNumber	int

As we saw for events, tools can introspect JavaBeans looking for methods that match a naming convention to infer properties. Many JavaBean tools will display the properties in some sort of a "property sheet" that allows a developer to view and modify the properties at design time.

This code snippet shows a fragment of a Bean class that defines three methods. The combination of the "getAddress" and "setAddress" methods implies a property named "address", while the existence of the "getSerialNumber" method implies a read-only property named "serialNumber".

Note that property names always begin with a lower-case character even though the method names include an upper-case character.

Also note that there's a special case for properties of type boolean: the "getter" method should be prefixed with "is" rather than "get", for example, "isHonorRoll()".

Firing an Event Based on Property Change

- It's common for a JavaBean to fire an event based on the value of a property change
- This is similar, but a bit different than a **bound** property which fires an event regardless of the property value

```
public void setGpa(double gpa)
{
   this.gpa = gpa;

   if (gpa < 1.0)
     fireFlunkedOutEvent ("Java 101");
}
```

Here we show an example JavaBean that fires the FlunkedOut event based on the GPA value, invoked the method shown a couple of pages ago.

We will cover bound properties in a moment, but note that this code only fires the event if the GPA is less than 1.0, while a bound property fires an event on any change to the property.

Indexed Properties

- **Indexed** properties are like simple properties accessed as an array

```
private String[] degrees = new String[4];

public String getDegree(int index)
{
    return degrees[index];
}
public void setDegree(int index, String degree)
{
    degrees[index] = degree;
}
```

Indexed properties are somewhat rarely used, but can be convenient. Here we show an indexed property named "degree" of type String. Note how the getter and setter methods accept the index.

In this example, we implemented the indexed property using an array, but that's not required (we could've used a collection class such as ArrayList instead). The important thing is that the get/set methods follow the naming pattern.

Also note that you can optionally provide methods that get/set the entire array, for example, String[] getDegrees().

Bound Properties

- A **bound** property fires the predefined **PropertyChange** event when the property changes to notify interested listeners
- The PropertyChangeEvent object encapsulates the property name and the previous and new values of the property
- The java.beans.PropertyChangeSupport class makes implementing bound properties easy

A bound property fires an even whenever the property changes. This is especially handy in GUI applications where you want a segment of the user interface to change when some other segment changes. For example, consider a pair of list boxes, one of which shows a list of managers, while the other shows the list of subordinates of the currently selected manager.

If a given JavaBean has multiple bound properties, it fires the same event whenever any property changes -- note that the PropertyChangeEvent object includes the property name -- that allows listeners to determine which property changed.

Also note that the PropertyChangeEvent object includes the previous property value -- that helps listeners to decide whether or not to take action based on the change.

Implementing a Bound Property

```
public class Student implements Serializable
{
    private String major;
    . . .
    private PropertyChangeSupport changes
       = new PropertyChangeSupport(this);

    public void setMajor(String major)
    {
        String priorMajor = this.major;
        this.major = major;
        changes.firePropertyChange("major",
            priorMajor, major);
    }
    public String getMajor()
    {
        return this.major;
    }
```

Here we show a Student JavaBean that provides a bound property named "major". Whenever the major changes, the bean fires the PropertyChange event.

Note that the code is fairly trivial since the java.beans.PropertyChangeSupport class does most of the work!

Implementing a Bound Property, cont'd

```
public void addPropertyChangeListener(
       PropertyChangeListener l)
    {
        changes.addPropertyChangeListener(l);
    }

    public void removePropertyChangeListener(
       PropertyChangeListener l)
    {
        changes.removePropertyChangeListener(l);
    }
. . .
}
```

Completing the implementation of the bound property, here we show the methods that allow listeners to register and de-register for the property-change event.

Listening to a Bound Property

```java
public class RespondToMajorChange
    implements PropertyChangeListener
{
    public void propertyChange(PropertyChangeEvent evt)
    {
        System.out.println("Property name: "
                + evt.getPropertyName());
        System.out.println("Old value: "
                + evt.getOldValue());
        System.out.println("New value: "
                + evt.getNewValue());
    }
    public static void main(String[] args)
    {
        RespondToMajorChange me =
           new RespondToMajorChange();
        Student s = new Student();
        s.addPropertyChangeListener(me);
        s.setMajor("Computer Science");
    }
}
```

Here we show a class that implements the PropertyChangeListener interface, registers and object and then simply prints the property name, old value and new value.

Advanced JavaBean Techniques

- **Constrained** properties are like bound properties, but any listener can veto the change
- You can override or supplement the standard reflection and introspection of a JavaBean by providing a BeanInfo class
- Providing customized GUIs and property editors for tools that manipulate JavaBeans
- Providing a customized, serialized instance of a JavaBean

Review Questions

- What differentiates a JavaBean from a normal Java class?
- What differentiates a JavaBean from an Enterprise JavaBean?
- What are the three kinds of features that a JavaBean can expose?

Chapter Summary

In this chapter you learned:

- Why JavaBeans are useful and sample applications of JavaBeans
- About the three JavaBean features: events, methods and properties