301AA - Advanced Programming

Lecturer: Andrea Corradini
andrea@di.unipi.it
http://pages.di.unipi.it/corradini/

AP-08: JavaBeans
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)


➔ The JavaBeans API Specification, sections 1, 2, 6, 7 and 8.
https://www.oracle.com/technetwork/java/javase/documentation/spec-136004.html
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
- Rich clients + application client components
- Web service clients

Web server tier
- JSP container + JSPs + Servlets
- Entity beans + Stateful & stateless session beans + Message-driven beans

App server tier
- EJB container

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 (e.g., a button in a GUI) to medium (e.g., a spreadsheet as part of a larger document) — Similar to Microsoft's 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` (Really needed?)
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
- Beans 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
public java.awt.Color getBackgroundColor ();
public 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 designers
• \(<\text{BeanName}>\text{BeanInfo}\) class to explicitly infer info on a bean
• Implicit method: based on reflection, naming conventions, and design patterns
Design Patterns in few slides

• A fundamental concept in Software Engineering & Programming, useful whenever one is designing a solution to a problem

• We shall meet several Design Patterns along the course (e.g., Observer or Publish-Subscribe, Visitor, Template Method,...)

• Just a brief introduction...
Design Patterns: From Architecture to Software Development

• Invented in the 1970's by architect Christopher Alexander:
  "Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice"
  
  Christopher Alexander, A Pattern Language, 1977

• The book includes 253 patterns for architectural design
• Common definition of a pattern:
  “A solution to a problem in a context.”
• Patterns can be applied to many different areas of human endeavour, including software development (where they are more successful!)
(Software) Design Patterns

• A (software) design pattern is a general, reusable solution to a commonly occurring problem within a given context in software design.

• Different abstraction levels:
  – Complex design for an entire application or subsystem
  – Solution to a general design problem in a particular context
  – Simple reusable design class such as a linked list, hash table, etc.
Patterns solve **software structural problems** like:

- Abstraction,
- Encapsulation
- Information hiding
- Separation of concerns
- Coupling and cohesion
- Separation of interface and implementation
- Single point of reference
- Divide and conquer
Patterns also solve **non-functional problems** like:

- Changeability
- Interoperability
- Efficiency
- Reliability
- Testability
- Reusability
Patter Template (an example)

• **Name**: meaningful text that reflects the problem, e.g. Bridge, Mediator, Flyweight

• **Problem addressed**: intent of the pattern, objectives achieved within certain constraints

• **Context**: circumstances under which it can occur, used to determine applicability

• **Forces**: constraints or issues that solution must address, forces may conflict!

• **Solution**: the static and dynamic relationships among the pattern components. Structure, participants, collaboration. Solution must resolve all forces!
## The 23 Design Patterns of the Gang of Four

Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides

*Design Patterns: Elements of Reusable Object-Oriented Software [1995]*

<table>
<thead>
<tr>
<th>Class</th>
<th>Behavioural</th>
<th>Creational</th>
<th>Structural</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>FM</em></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Factory Method</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><em>PT</em></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Prototype</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><em>S</em></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Singleton</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><em>AF</em></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Abstract Factory</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><em>TM</em></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Template Method</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><em>CD</em></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Command</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><em>MD</em></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mediator</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><em>O</em></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Observer</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><em>IN</em></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Interpreter</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><em>CR</em></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Chain of Responsibility</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><em>CP</em></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Composite</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><em>D</em></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Decorator</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><em>BU</em></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Builder</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><em>SR</em></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Strategy</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><em>MM</em></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Memento</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><em>ST</em></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>State</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><em>IT</em></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Iterator</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><em>V</em></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Visitor</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><em>FL</em></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Flyweight</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><em>BR</em></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bridge</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
A Sample Pattern: **Singleton**
(Creational)

**Name:** Singleton

**Problem:**
How can we guarantee that one and only one instance of a class can be created?

**Context:** In some applications it is important to have exactly one instance of a class, e.g. sales of one company.
**Forces:** Can make an object globally accessible as a global variable, but this violates encapsulation. Could use class (static) operations and attributes, but polymorphic redefinition is not always possible.

**Solution:**
- Create a class with a class operation `getInstance()`.
- When class is first accessed, this creates relevant object instance and returns object identity to client.
- On subsequent calls of `getInstance()`, no new instance is created, but identity of existing object is returned.
Singleton Structure

Singleton

- uniqueInstance
- singletonData

+ getInstance()
+ getSingletonData()
+ singletonOperation()
- Singleton()

getInstance() {
    if ( uniqueInstance == null )
        { uniqueInstance = new Singleton() }
    return uniqueInstance
}
Example: Code

Class Singleton {
    private static Singleton uniqueInstance = null;
    private Singleton( ) { .. } // private constructor
    public static Singleton getInstance( ) {
        if (uniqueInstance == null)
            uniqueInstance = new Singleton();
        // call constructor
        return uniqueInstance;
    }
}
Comments

• To specify a class has only one instance, we make it inherit from **Singleton**.

+ controlled access to single object instance through Singleton encapsulation

+ Can tailor for any finite number of instances

+ namespace not extended by global variables

- access requires additional message passing

- Pattern limits flexibility, significant redesign if singleton class later gets many instances
Back to JavaBeans: 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`

• Example:

```java
public java.awt.Color getBackground ();
public void setBackground (java.awt.Color color);
```

• 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

```java
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 discuss 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
Pattern: **Observer** (Behavioral) aka **Publish-Subscribe**

**Name:** Observer

**Problem:** Define a one-to-many dependency among objects so that when one object changes state, all of its dependents are notified and updated automatically.
Forms of design-level reuse

The catalog by Gamma et al. that is particularly close to many examples in this book is the Observer pattern. It defines a one-to-many dependency between objects. This is that when the one object (the subject) changes, all its dependents (the observers) are notified to perform updates as required. Using the notation of Gamma et al., Figure 9.1 shows the class diagram of the Observer pattern. The attached "notes" sketch implementations where this helps to understand the pattern. Otherwise, the notation is close to UML.

The idea is that, once the problem has been isolated, a proper pattern can be chosen. The pattern then needs to be adapted to the specific circumstances. For example, an observable subject may need to be observed by $n$ observers which are themselves unknown to the subject. The Observer pattern is chosen, but the pattern's Update method may need an additional argument to inform observers of what it is that has changed. A pattern catalog should contain a discussion of the common variations on a patterns theme, as demonstrated by Gamma et al.

Design patterns are microarchitectures. They describe the abstract interaction between objects collaborating to solve a particular problem. They are quite different from frameworks (described in the next section). Gamma et al. list the following differences between patterns and frameworks (Gamma et al., 1995, p. 28):

- **Subject**
  - +Attach(in Observer)
  - +Detach(in Observer)
  - +Notify()

- **Observer**
  - +Update()

- **ConcreteSubject**
  - -subjectState
  - +GetState()
  - +SetState()

- **ConcreteObserver**
  - -observerState
  - +Update()

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

Return subjectState

observerState = subject->GetState()
Events

• In Java the Observer pattern is 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: several possible listeners

• Unicast semantics (at most one listener) can be enforced by tagging the event source.
Design Pattern for Events

Based on methods for (un)registering listeners. From

```java
public void add<EventListType>(<EventListType> a)
public void remove<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);
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

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.

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

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

void doIt(FooEvent 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
// Adaptor to map "OK Pressed" events onto doOKAction()
class OKButtonAdaptor implements PushButtonExampleListener {
    private Dialog dialog;
    public OKButtonAdaptor(Dialog dest) {
        dialog = dest;
    }
    public void buttonPushed(PushButtonExampleEvent pbe) {
        dialog.doOKAction();
    }
}
```

okButton.addPBListener(okButtonAdaptor)
cancelButton.addPBListener(cancelButtonAdaptor)
```
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
   ```java
   private PropertyChangeSupport changes = new PropertyChangeSupport(this);
   ```
3. Implement methods to maintain the property change listener list:
   ```java
   public void addPropertyChangeListener(PropertyChangeListener l) {
     changes.addPropertyChangeListener(l);
   }
   (also `removePropertyChangeListener` method is needed)
Implement Bound Property in a Bean (cont.)

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 interface

   ```java
   public class MyLstnr implements PropertyChangeListener
   ```

2. It must override the method

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

3. Sample registration:

   ```java
   Button button = new OurButton();
   MyLstnr lis = new MyLstnr();
   button.addPropertyChangeListener(lis);
   ```
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 by raising an exception.
- 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. The change is rejected by raising a `PropertyVetoException`

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

The bean containing the constrained property must:

1. Import the `java.beans` package
2. Instantiate a `VetoableChangeSupport` object:
   ```java
   private VetoableChangeSupport vetos =
   new VetoableChangeSupport(this);
   ```
3. Implement methods to maintain the listener list:
   ```java
   public void addVetoableChangeListener(VetoableChangeListener l)
   {   vetos.addVetoableChangeListener(l);
   }
   ```
4. and similarly for `removeVetoableChangeListener`
Implement Constrained Property in a Bean (cont.)

5. Write a property’s setter method to fire a property change event:

```java
public void setX(int newX)
{
    int oldX = X;
    try{
        vetos.fireVetoableChange("X", oldX, newX);
        // if no veto there
        X = newX;
        // add here code to notify change, if needed
    } catch(PropertyVetoException e){
        // code to be executed if change is rejected by somebody
    }
}
```
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 the `vetoableChangeSupport` object.

3. If the listener wants to forbid the change described in `evt`, it should raise a `PropertyVetoException`. Otherwise simply return.
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 the JavaBean API