

# 301AA - Advanced Programming

Lecturer: **Andrea Corradini**

[andrea@di.unipi.it](mailto:andrea@di.unipi.it)

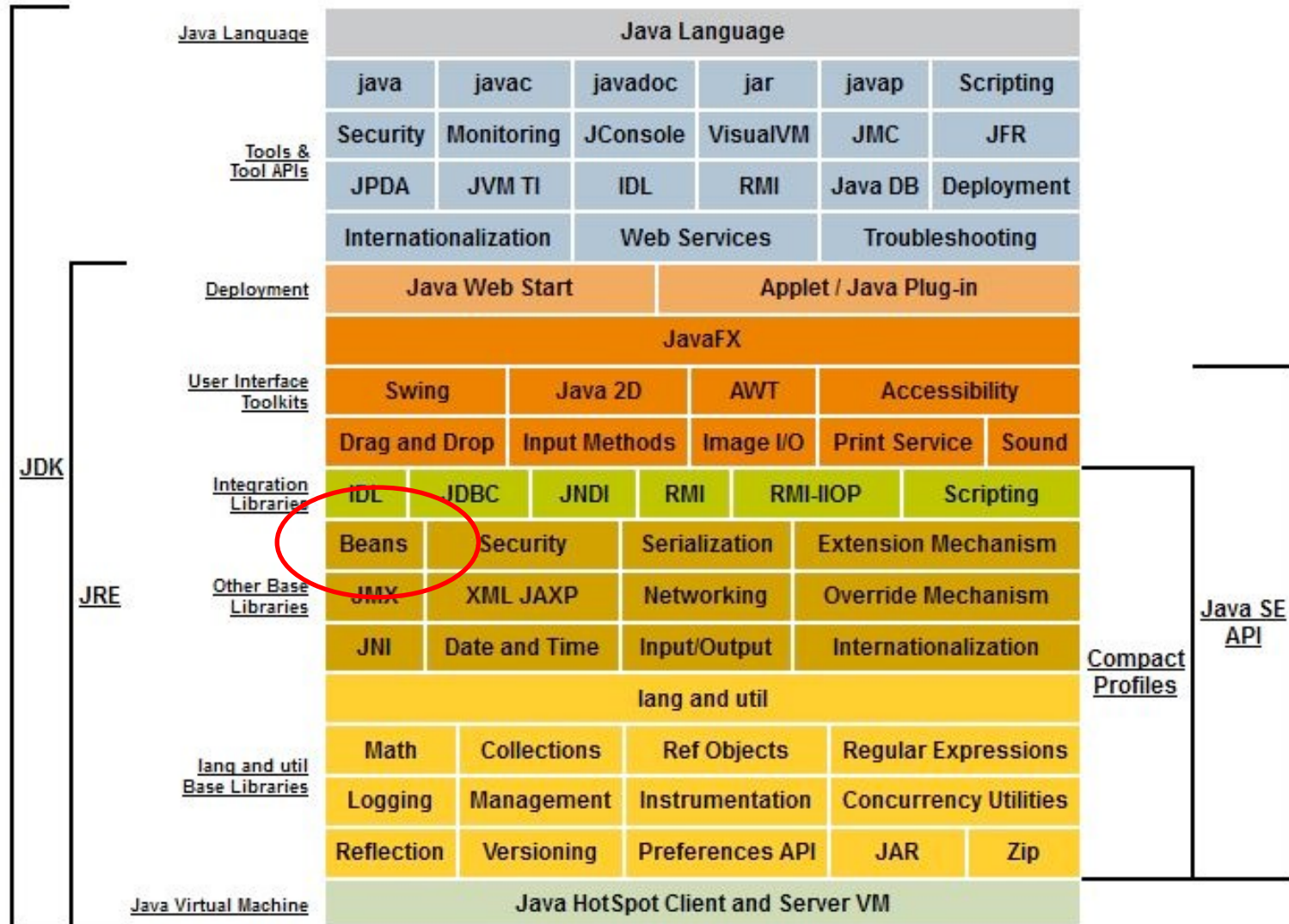
<http://pages.di.unipi.it/corradini/>

*AP-07: 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)
- ➔ Chapter 14, sections 14.1, 14.3 and 14.5 of *Component Software: Beyond Object-Oriented Programming*. C. Szyperski, D. Gruntz, S. Murer, Addison-Wesley, 2002.
- ➔ [The JavaBeans API Specification](https://www.oracle.com/technetwork/java/javase/documentation/spec-136004.html), 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/Jakarta EE (Enterprise Edition)
  - Suite of specifications for referencing runtimes (application servers / microservices)
  - Example of uses: e-commerce, banking information systems, accounting.
  - Around 10 referencing runtimes 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 Jakarta EE

## Client side

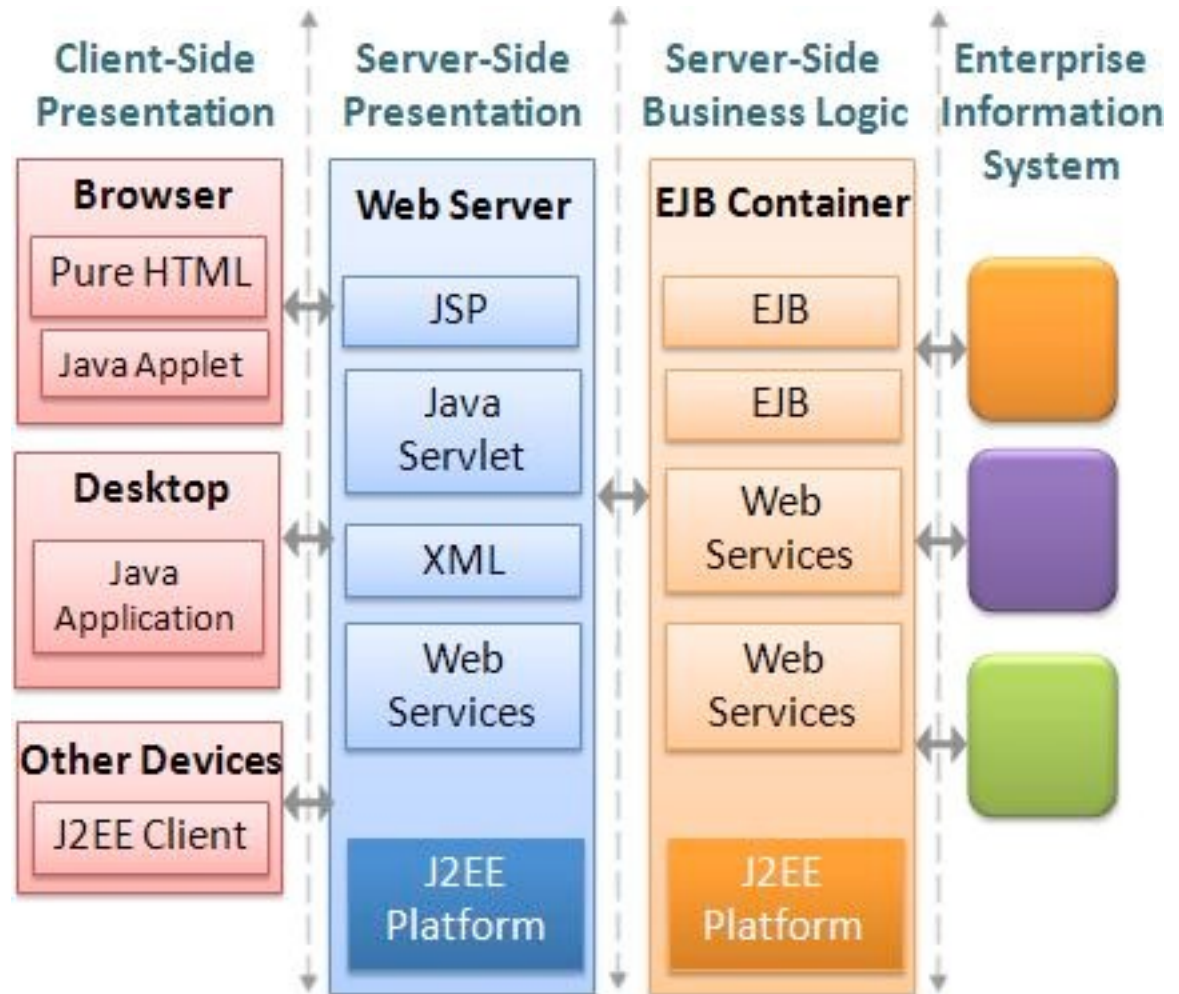
- JavaBeans
- Applets
- Application Components

## Web server tier

- Servlets
- JSPs

## Application tier:

- Stateless session EJB
- Stateful session EJB
- Entity EJB
- Message-driven EJB



# Components in Jakarta EE

## Client side

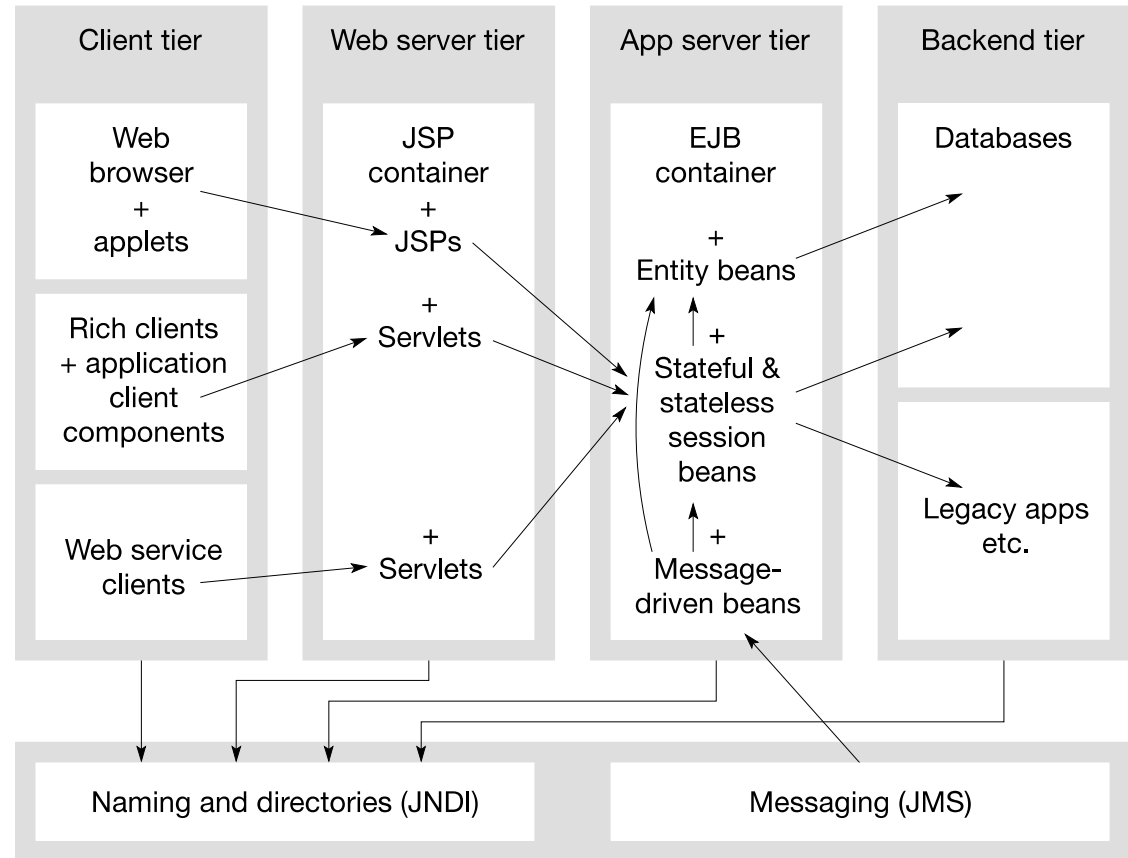
- JavaBeans
- Applets
- Application Components

## Web server tier

- Servlets
- JSPs

## Application tier:

- Stateless session EJB
- Stateful session EJB
- Entity EJB
- Message-driven EJB



# 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 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
  - It has a public constructor with no arguments
  - It implements the interface `java.io.Serializable`
  - It is in a **jar** file with *manifest file* containing  
Java-Bean: True (Really needed?)

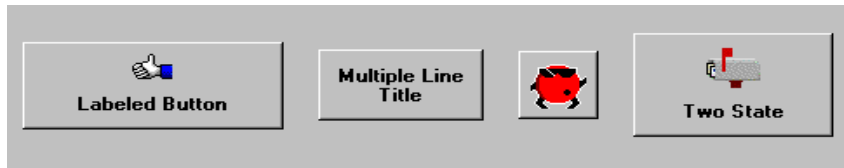


# JavaBeans as Software Components

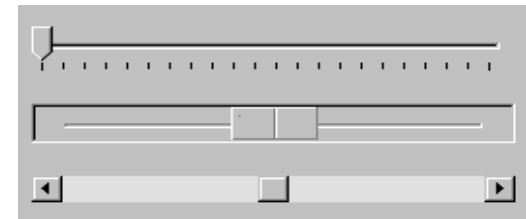
- Beans are binary building blocks (class files)
- Development vs. deployment (customization)
- Beans can be assembled to build a new bean or a new application, applet, ... writing glue code to wire beans together
- Client-side beans vs. beans for business logic process in MVC on server
- Beans on server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

# Sample Reusable Components

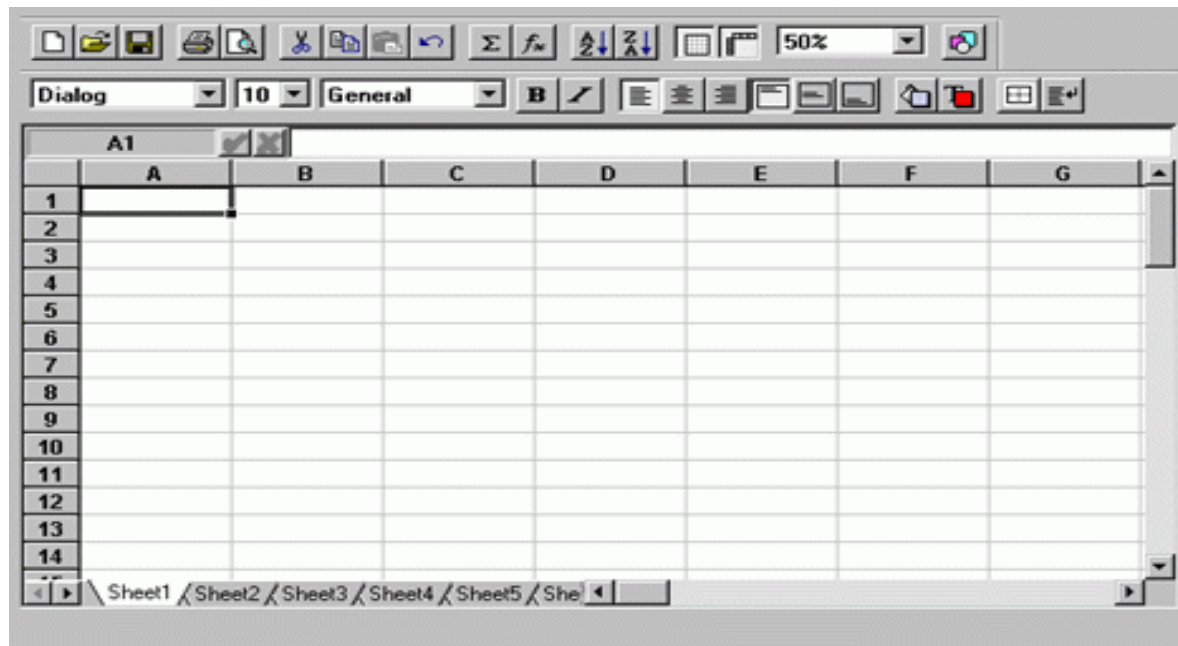
## Button Beans



## 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 from 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 of a button: **background**

```
public java.awt.Color getBackground ();  
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 capabilities
- Allows application builder tool to present info about a component to software designers
- Implicit method: based on **reflection, naming conventions, and design patterns**
- Alternative: *<BeanName>BeanInfo* class to explicitly describe info about a bean for the builder tool

# Using the BeanInfo class

With the BeanInfo class you can:

- Expose only those features you want to expose.
- Rely on BeanInfo to expose some Bean features while relying on low-level reflection to expose others.
- Associate an icon with the target Bean.
- Specify a **customizer** class.
- Segregate features into normal and expert categories.
- Provide a more descriptive display name, or additional information about a Bean feature.



# Design Pattern for Simple Properties

- From pair of methods:

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

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

- Example:

```
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

```
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**

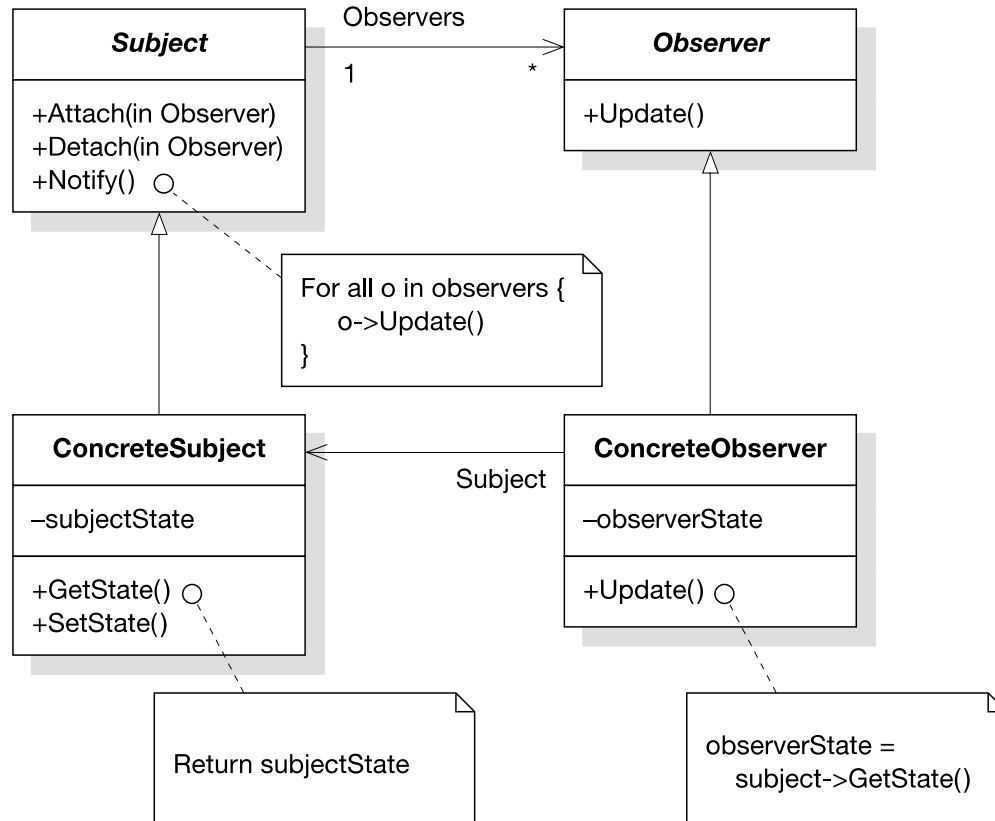
# Connection-oriented programming

- Paradigm for gluing together components in a builder tool
- Loose coupling
- 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.



# 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

```
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

```
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:

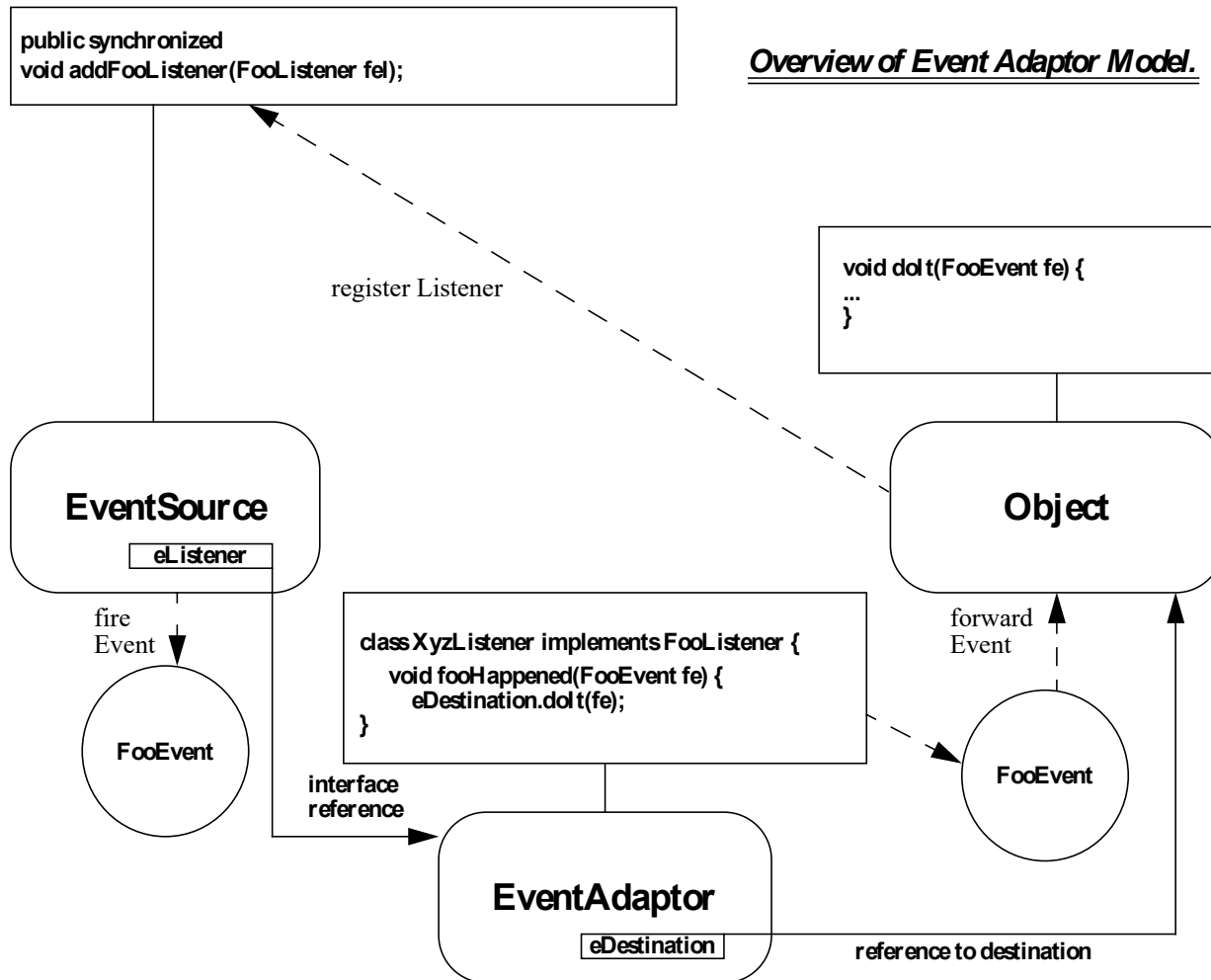
```
public void addJackListener(JackListener t)
throws java.util.TooManyListenersException;
public void removeJackListener(JackListener t);
```

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

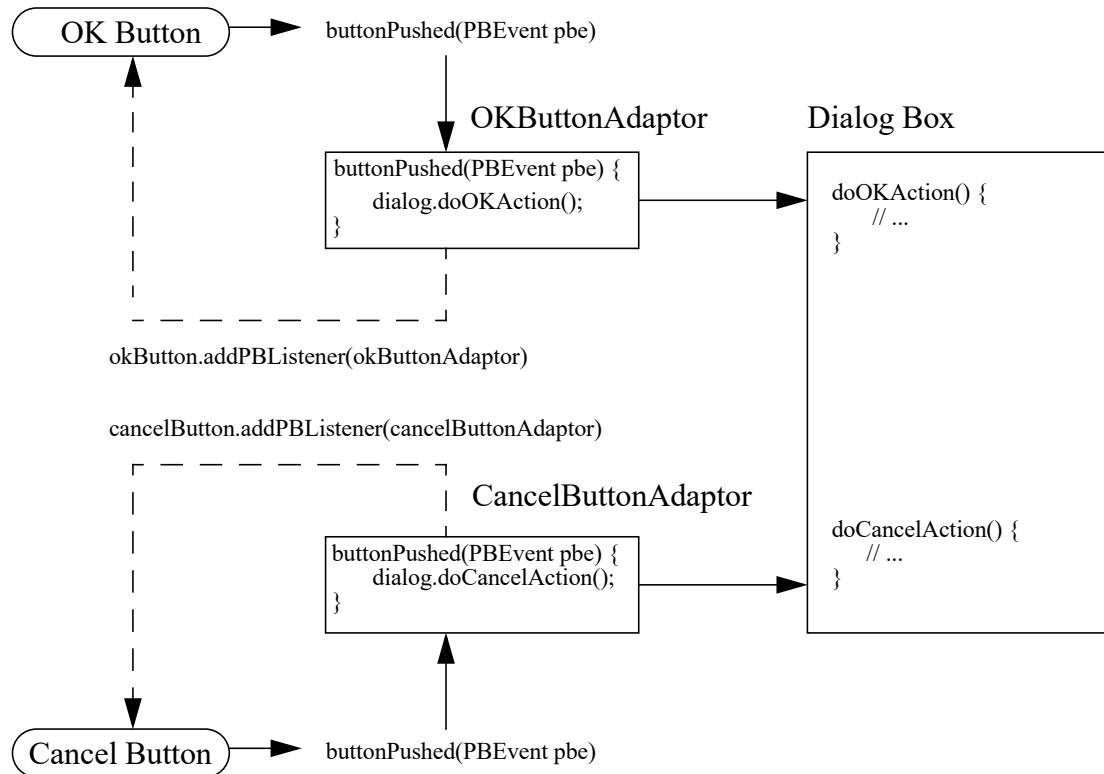
# Event Adaptors

- 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: general architecture



# Event adaptors example: Demultiplexing multiple event sources



# Back to Bound Properties

- 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 (cont.)

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

```
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 **PropertyChangeListener**

```
public class MyLstnr implements  
    PropertyChangeListener, Serializable
```

2. It must override the method

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

3. Sample registration:

```
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 for 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. These objects either accept or reject 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:

```
private VetoableChangeSupport vetos =  
    new VetoableChangeSupport(this);
```

3. Implement methods to maintain the listener list:

```
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:

```
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  
**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