

JavaBeans

Properties und Events

© J. Heinzlreiter
WS 2004/05

JavaBeans - Literatur

- Bücher

- L. Vanhelsuwé: *Mastering JavaBeans*
- R. Leinecker et al: *JavaBeans Unleashed*
- Bruce Eckel: *Thinking in Java*

- Tutorial

- <http://java.sun.com/docs/books/tutorial/javabeans/index.html>

- Links

- <http://java.sun.com/beans>
- <http://java.sun.com/products/javabeans/docs/spec.html>

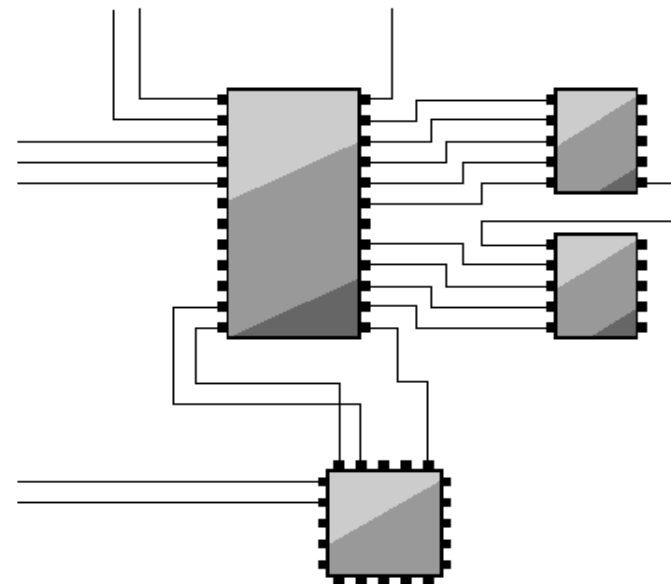
JavaBeans - Definition

- *JavaBeans* sind eine Spezifikation von Sun.
- „Ein *JavaBean* ist eine Plattform-unabhängige Softwarekomponente, die mit einem Builder Tool manipuliert werden kann.“
- Ziel: Definition eines Komponentenmodells für Java.
- Abgrenzung zu ActiveX-Controls
 - JavaBeans sind Plattform-unabhängig, funktionieren aber nur in Java-Laufzeitumgebung.
 - ActiveX-Controls sind sprachunabhängig, sind aber nur auf Windows-Plattformen einsetzbar.

Was sind JavaBeans?

- JavaBeans treten mit Umgebung nur durch ihr *Interface* in Interaktion (Black Box).

- Interface besteht aus 3 Komponenten:
 - Methoden,
 - Properties,
 - Events.



- JavaBeans können zur Designzeit von einem Tool manipuliert werden („Tool-Awareness“).

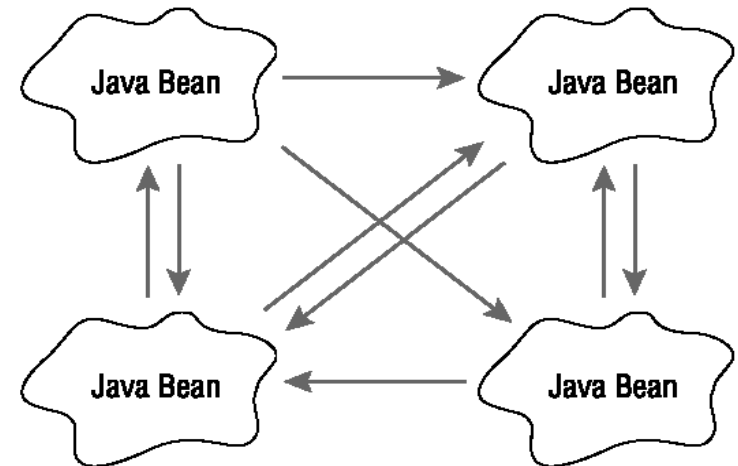
JavaBeans – Properties und Events

- Properties

- Definiert das Verhalten und Aussehen der Komponente.
- Änderung einer Property wird sofort sichtbar.

- Events

- Viele kleine Komponenten
- Interaktion:
 - Feuern von Events,
 - Empfangen von Events.
- Events haben die Funktion der Pins bei ICs.



JavaBeans - Environments

- Design-Time Environment
 - Bean „lebt“ bereits im Builder-Tool (Bean wird zur im Builder-Tool instanziiert).
 - Objekte in Frameworks sind zur Design-Zeit noch „tot“.
 - Look&Feel der Komponente ist schon bekannt.
- Run-Time Environment
 - JavaBean läuft in einer Java Virtual Machine.
 - JavaBeans sind *keine verteilbaren* Komponenten (im Gegensatz zu CORBA-Komponenten).
 - JavaBeans leben in einem „*multithreaded* Environment“.

Bean Properties

- Anwendung
 - Manipulationsmöglichkeit mit *Builder Tool*.
 - Ändern der Properties mit *Property Editor*.
 - Änderung einer Property wirkt bereits zur *Designzeit* auf visuelle Repräsentierung.
- Arten von Properties
 - Simple Properties,
 - Bound Properties,
 - Indexed Properties,
 - Constrained Properties.

Realisierung von Properties

- Öffentliche Datenkomponenten werden nicht als Properties exportiert.
- Properties werden über *Zugriffs-Methoden* realisiert.
 - Lesen der Property (*Getter*)
 - Schreiben der Property (*Setter*)
- Exportieren der Properties
 - Namenskonventionen
 - Klasse hinzufügen, die von BeanInfo abgeleitet ist.

Simple Properties (1)

- *Read Property*

- Syntax:

- publ i c PropertyType getPropertyName ()

- Beispiel:

- *Richtig:* publ i c Color getBackgroundCol or()

- *Falsch:* publ i c Color getBackgroundCol or(int i)

- *Write Property*

- Syntax:

- publ i c void setName (PropertyType name)

- Beispiel:

- *Richtig:* publ i c void setBackgroundCol or(Color c)

- *Falsch:* publ i c void setPosition(int x, int y)

Simple Properties/Boolean Properties

- *Read/Write Property*

- Property-Name in Getter- und Setter-Methode muss übereinstimmen.

- Beispiel:

- Folgende zwei Methoden würden zwei Properties definieren:

```
public Price getDiscountedValue()
```

```
public void setDiscountValue(Price value)
```

- *Boolean Property*

- Syntax:

```
public boolean isPropertyName ()
```

- Beispiel:

- public boolean isValid()

- void setValid(boolean v)

Indexed Properties

- *Arrays* erfordern keine spezielle Behandlung:
 - `public PropertyType[] getPropertyNames()`
 - `public void setPropertyNames(PropertyType[] array)`
- Mit *Indexed Properties* können einzelne Array-Elemente angesprochen werden:
 - Syntax:
 - `public PropertyType getPropertyNames (int index)`
 - `public void setPropertyNames (int index, PropertyType value)`

Bound Properties

- Merkmale
 - Eigenschaften von Simple Properties.
 - Änderung der Property wird registrierten Clients mitgeteilt.
- Anwendungsbeispiele
 - Spreadsheet: Änderung in einer Zelle wirkt sich sofort auf Chart-Bean aus.
 - Visuelles Programmierwerkzeug: Änderung einer Property bewirkt Änderung im Source-Code
- Vorteil:
 - Beans können lose gekoppelt werden.
 - Listener sind anonym.

Bound Properties – Verwendung

- Anmeldung/Abmeldung

- bean. `addPropertyChangeListener` (propChangeListener)
- bean. `removePropertyChangeListener` (propChangeListener)

- Interface

```
public interface PropertyChangeListener
    extends EventListener {
    void propertyChange(PropertyChangeEvent e);
}
```

- PropertyChangeEvent

```
public class PropertyChangeEvent extends EventObject {
    public PropertyChangeEvent(Object src,
        String propName, Object oldValue, Object newValue);
    public String getPropertyName();
    public Object getNewValue();
    public Object getOldValue();
}
```

Bound Properties – Beispiel (1)

```
public class Client implements
    PropertyChangeListener {
    protected Clock clock;

    public Client {
        clock = new Clock();
        clock.addPropertyChangeListener (
            this);
    }
}
```

```
public void propertyChange (
    PropertyChangeEvent pcEvent) {
    String changedProp =
        pcEvent.getPropertyName();
    Object oldPropValue =
        pcEvent.getOldValue().;
    Object newPropValue =
        pcEvent.getNewValue();

    System.out.println(changedProp);
    ...
}
```

Bound Properties - Implementierung

- Hilfsklasse `PropertyChangeSupport`

```
public class PropertyChangeSupport ... {  
    public PropertyChangeSupport(Object bean);  
    public synchronized void addPropertyChangeListener  
        (PropertyChangeListener l);  
    public synchronized void removePropertyChangeListener  
        (PropertyChangeListener l);  
    public void firePropertyChange(  
        String propName, Object oldValue, Object newValue);  
}
```

- Bean kann von Hilfsklasse abgeleitet werden.
- Bean kann Hilfsklasse enthalten (Komposition).

Bound Properties – Beispiel (2)

```
public class Clock {  
    protected int tickInterval;  
    protected PropertyChangeSupport  
        changer = new  
            PropertyChangeSupport(this);  
  
    public int getTickInterval() {  
        return tickInterval;  
    }  
  
    public setTickInterval(int newInterval) {  
        int oldInterval = tickInterval;  
        tickInterval = newInterval;  
        changer.firePropertyChange(  
            "tickInterval",  
            new Integer(oldInterval),  
            new Integer(newInterval));  
    }  
}
```

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


Per-Property Listener

- Client wird von Änderungen in jeder Bound Property informiert.
- Listener können auch pro Bound Property definiert werden:
 - void `addPropertyNameListener`
 (`PropertyNameListener` l);
 - void `removePropertyNameListener`
 (`PropertyNameListener` l);

Constrained Properties

- Merkmale

- Teilen wie Bound Properties registrierten Listenern Wertänderungen mit.
- *Wertänderung* kann *verweigert* werden.
- Nicht das Bean, sondern ein *Listener* kann die Wertänderung verweigern.
- Verweigerung der Wertänderung wird mit *Exception* mitgeteilt.

- Probleme

- Andere Listener können bereits von Wertänderung informiert sein, wenn Wertänderung zurückgewiesen wird.

Events

- Anwendung
 - Kommunikation mit der Außenwelt,
 - Koppelung verschiedener Komponenten.
- Implementierung
 1. Definition eines Event-Objekts
 - Dient zum Transport von Event-Parametern.
 - Unterklasse von `java.util.EventObject`
 2. Definition eines Event-Interfaces.
 - Clients, die von Events verständigt werden wollen, müssen dieses Interface Implementieren.
 - Abgeleitet von `java.util.EventListener`
 3. Definition eines Adapters (optional).
 - Leer-Implementierung des Event-Interfaces.
 - Client muss nicht alle Methoden des Interfaces implementieren.
 4. Erweiterung der Event-Quelle.

Implementierung des Event-Objekts (1)

- Ableiten von `java.util.EventObject`

```
public class MyEvent extends EventObject {
    protected Attribute1 a1;
    public MyEvent(Object source, Attribute1 a1) {
        super(source);
        this.a1 = a1;
    }
    public int getAttribute1() { return a1; }
    ...
}
```

Event-Objekt – Beispiel

```
public class ClockEvent extends EventObject {
    protected long timeStamp;
    public ClockEvent(Object source, long timeStamp) {
        super(source);
        this.timeStamp = timeStamp;
    }
    public long getTimeStamp() {
        return timeStamp;
    }
}
```

Impl. des Interfaces und des Adapters (2+3)

- Interface (2)

- Erweitern von `java.util.EventListener`

```
public interface MyBeanListener
    extends EventListener {
    void myEventOccurred(MyEvent e);
    ...
}
```

- Adapter (3)

- Alle Methoden eines Interfaces müssen definiert werden.
- Adapter ist eine Standard-Implementierung eines Interfaces

```
public class MyBeanAdapter implements MyBeanListener {
    public void myEventOccurred(MyEvent e) {}
}
```

Event-Interface – Beispiel

- Interface

```
public interface ClockListener extends EventListener {  
    void clockTicked(ClockEvent e);  
    void timerExpired(ClockEvent e);  
}
```

- Adapter

```
public class ClockAdapter implements ClockListener {  
    public void clockTicked(ClockEvent e) {}  
    public void timerExpired(ClockEvent e) {}  
}
```

Erweiterung der Event-Quelle (4)

- Methoden zum Hinzufügen und Entfernen von Listenern definieren:

```
public void addMyBeanListener(  
    MyBeanListener listener);
```

```
public void removeMyBeanListener(  
    MyBeanListener listener);
```

- Container für Listener-Objekte (Vector, ArrayList) zum Bean hinzufügen.
- Feuern des Events:

```
e = new MyEvent(this, eventArgs)  
Iterator it = listeners.iterator();  
while (it.hasNext())  
    ((MyBeanListener)it.next()).myEventOccurred(e)
```


Impl. einer Eventquelle – Beispiel

```
public class Clock {
    protected Vector clockListeners;
    protected long timeStamp;
    public Clock(long timeStamp) {
        this.timeStamp = timeStamp;
        clockListeners = new Vector();
    }
    public void addClockListener (
        ClockListener l) {
        clockListeners.add (l);
    }
    public void removeClockListener(
        ClockListener l) {
        if (! clockListeners.remove(l))
            throw new
                IllegalArgumentException("...");
    }
}
```

```
protected void fireTickEvent() {
    ClockEvent event = new ClockEvent(
        this, timeStamp);
    Vector listeners =
        (Vector)clockListeners.clone();
    Iterator l = listeners.iterator();
    while (l.hasNext()) {
        ((ClockListener)l.next()).
            clockTicked(event);
    }
}
```

Implementierung des Clients

- Schritte

- Client implementiert Event-Listener-Interface.
- Client registriert sich bei der Event-Quelle.

```
public class MyClient implements MyBeanListener {
    protected MyBean bean;

    public MyClient() {
        bean = new MyBean();
        bean.addMyBeanListener(this);
    }

    public dispose() {
        bean.removeMyBeanListener(this);
    }

    public void myEventOccurred(MyEvent e) { ... }
}
```

Event-Client – Beispiel

```
public class ClockClient extends Frame implements ClockListener {
    Clock clock;

    public ClockClient() {
        clock = new Clock(currentTime);
        clock.addClockListener(this);
        addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                clock.removeClockListener(this);
                System.exit(0); });
    }

    public void clockTicked(ClockEvent e) {
        System.out.println("Clock tick: " + e.getTimestamp());
    }

    public void timerExpired(ClockEvent e) {
        System.out.println("Timer event: " + e.getTimestamp());
    }
}
```

Bean Events – Namenskonventionen

- Exportieren von Events

- Namenskonventionen
- BeanInfo

- Namenskonventionen

- public void **addEventTypeListener**(**EventTypeListener** l);
- public void **removeEventTypeListener**(**EventTypeListener** l);
- Beispiele:
 - *Richtig*: public void **addTimeOutListener**(**TimeOutListener** l);
 - *Falsch*: public void **addColorListener**(**Color** l);
 - *Falsch*: public int **addActionListener**(**ActionListener** l);

Java Eventmodell: Zusammenfassung

