JAVABEANS ÜBERSICHT

In diesem Kapitel:

JavaBeans

-

Übersicht

1.1. Einleitung

1.1.1. Übergeordnete Kursziele

Das übergeordnete Ziel dieser Einführung in die JavaBeans Technologie ist es, Ihnen zusammen mit praktischen Beispielen oder Übungen folgende Technologien näher zu bringen:

- die JavaBeans Architekture
- das Beans Event Modell
- Introspektion - Abfrage des Inhaltes einer JavaBean
- kreieren von Bean Komponenten
- Anpassen von Beans
- Persistenz zum Speichern und Aktivieren von Beans
- die BDK BeanBox

1.1.1.1. Kurs-Voraussetzungen

Dieser Kurs setzt voraus, dass Sie mit den Objekt-orientierten Konzepten vertraut sind und AWT, Threads und natürlich die Java Grundlagen kennen.

1.1.1.2. Format des Kurses

Dieser Kurs besteht aus Übungen und etwas Theorie. Vorallem sollten Sie aber einen praktischen Einblick in die JavaBeans gewinnen. Dies ist nur möglich, falls Sie die Übungen mindestens anschauen. Alle Übungen enthalten auch Musterlösungen, so dass Sie mindestens eine Ahnung über die praktische Umsetzung erhalten.

1.1.1.3. Vorgehensweise beim Durcharbeiten

Jede Person hat ihren eigenen Lernstil. Sie können mit den Übungen beginnen und dann zur Theorie wechseln. Sie können aber auch in diesem Theorieteil starten und dann zu den Übungen übergehen.

1.1.2. Lernziele

In dieser Kurseinheit lernen Sie

- die JavaBeans Architektur
- das Beans Event Modell
- Introspektion zur Abfrage des Inhalts der Beans
- das Kreieren der Bean Komponenten
- Customizing / Anpassungen an Beans
- Persistenz (dauerhafte Speicherung und Wiedergewinnung) von Beans

- Entwickeln von Applikationen, welche Bean Komponenten umfassen
- Entwickeln einfacher Applikationen, welche Beans verwenden
- das Bean Development Kit (BDK) / Bean Box

kennen.

Erklärtes Ziel ist es, dass Sie lernen, wie man JavaBeans effektiv entwickelt. Dazu gehen wir grob folgendermassen vor:

1) erklären der Services, welche benötigt werden, um JavaBeans einzusetzen:
   Reflection / Introspection;
   Event Handling / Kommunikation;
   Customizing / GUI Builder Support;
   Persistenz / Serialisierung;
   GUI Merging / Properties
2) nachdem Sie dieses Hintergrundwissen besitzen, lernen Sie Customizing, Kommunikation mit JavaBeans und Integration
3) schliesslich lernen Sie wie man selber eigene JavaBeans schreibt.

Soweit das Programm dieser Kurseinheit.

1.2. Einführung in JavaBeans

JavaBeans versuchen den Slogan "Write once, Run anywhere" zu erweitern und zwar durch "reuse everywhere". JavaBeans sind plattformunabhängige Software Komponenten, in Java geschrieben. Sie können damit kleine, wiederverwendbare Softwarekomponenten schreiben. Ein Builder Programm kombiniert diese Komponenten aus unterschiedlichen Quellen, um Applikationen zu kreieren.

1.2.1. Was ist eine Bean?

Eine Bean ist eine Java Komponente. beans sind unabhängige, wiederverwendbare Softwaremodule. Beans können sichtbare Objekte, wie beispielsweise AWT Komponenten, oder unsichtbare Objekte, wie Stacks, Listen, Warteschlangen.... sein. Ein Builder Tool erlaubt die Integration unterschiedlicher Beans, um daraus Applikationen und Applets zu bauen.

1.2.1.1. Übung

Setzen Sie das bean Development Kit auf und entwickeln Sie / modifizieren Sie Ihre erste JavaBean.

1.2.2. Beans Architekture

Beans bestehen aus drei Teilen:

- Events
- Properties
- Methoden

Da Beans zustandsabhängig sind, benötigen wir zudem eine Möglichkeit diese Zustände persistent, also längerfristig zu speichern, so dass der Zustand auch nach dem Herunterfahren und erneuten Starten des Rechners noch verfügbar ist.

1.2.2.1. Events

Events sind nichts anderes als irgend etwas, was geschieht, was passieren kann. Das Delegations-Event Modell wurde in Java im AWT ab version Java 1.1 eingeführt. Dieses Event Modell entspricht genau dem Beans Event Modell. Es besteht aus drei Teilen:

- `EventObject`
  Ein AWT Event in der AWT Welt
- `EventListener`
  `ActionListener, ItemListener`, ...
- Event Source (Beans)
  unterschiedliche AWT Komponenten.

Jedermann kann einen `EventListener` mit einer `Component` registrieren, sofern die Component das Event (welches gesetzt wird) interpretieren kann. Sie können beispielsweise keinen `ActionListener` mit einer `TextArea` kombinieren, wohl aber mit einem `TextField`. Falls mit der Komponente, dem `Component` Objekt, etwas passiert, wird der/die Listener informiert, indem ein `EventObject` mit den passenden Listenermethoden versendet wird.

1.2.2.2. Properties

Properties definieren die Charakteristiken der Bean. Zum Beispiel besitzt ein AWT Textfeld mehrere Eigenschaften, wie den aktuellen Text, Echo Charakter und vieles mehr. Im einfachsten Fall wird eine Eigenschaft mit einer Methode / Methoden in folgendem Design Pattern definiert:

```
public void set PropertyName(PropertyType value);
public PropertyType getPropertyName();
```

`PropertyName` ist der Name der Eigenschaft, `PropertyType` der dazugehörige Datentyp.Falls nur eine Methode vorhanden ist, ist die Eigenschaft entweder read-only (die `set..()`. Methode fehlt) oder write-only (die `get...()` Methode fehlt).

1.2.2.3. Methoden

Die Bean-Methoden stehen jedermann mittels öffentlichen Methoden zur Verfügung. Sie können festlegen, welche Methoden dem Builder / Integration Tool bekannt gegeben werden. Dazu verwenden Sie die `getMethodDescriptors()` Methode zusammen mit `BeanInfo`.Jede Bean kann eine `BeanInfo` Klasse zur Verfügung stellen, mit der das Erscheinungsbild der Bean dem Builder Tool bekanntgegeben wird. Weiter unten finden Sie mehr Informationen zu dieser Klasse.

1.2.2.4. Persistenz

Persistenz beschreibt die Fähigkeit eines Objekts seinen Zustand abzuspeichern, um später wieder zulesen. Beans benutzen die Objekt-Serialisierung zur Realisierung der Persistenz. Die zwei Interfaces `ObjectInput` und `ObjectOutput` bilden die Basis für die Serialisierung in Java.

Diese Interfaces werden mit den Klassen `ObjectInputStream` und `ObjectOutputStream` implementiert. Die Serialisierung speichert alle nicht-statischen, nicht-transienten Instanzenvariablen eines Objekts. Falls ein Objekt ein weiteres Objekt referenziert, wird auch dieses rekursive gespeichert, serialisiert. Falls ein Objekt mehrfach referenziert wird, wird es lediglich einmal serialisiert. Damit wird die Konsistenz der Objekte garantiert.

Hier ein einfaches Beispiel:
Zuerst die Objekte:

```
TreeNode top = new TreeNode("top");
top.addChild(new TreeNode("left child"));
top.addChild(new TreeNode("right child"));
```

und dann die Serialisierung:

```
FileOutputStream fOut =
  new FileOutputStream("test.out");
ObjectOutput out = new ObjectOutputStream(fOut);
out.writeObject(top);
out.flush();
out.close();
```

Den Zustand der serialisierten Objekte können Sie aus den gespeicherten (serialisierten) Objekten folgendermassen zurück gewinnen:

```
FileInputStream fIn = new FileInputStream("test.out");
ObjectInputStream in = new ObjectInputStream(fIn);
TreeNode n = (TreeNode)in.readObject();
```

1.2.2.5. Technologievergleich mit ActiveX/COM

In der literatur finden Sie verschiedene vergleiche der Komponenten-technologien von Microsoft und Java. Hier einige Punkte:

- JavaBeans stellen ein Framework zur Entwicklung komponentenbasierter Anwendungen in Java zur verfügung.
- ActiveX ist ein Framework zur Entwicklung von Compound Document Anwendungen, mit ActiveX Controls.
- Ein Bean entspricht in etwa einer ActiveX Control. Allerdings sind Beans in Java geschrieben und garantieren eine plattformübergreifende Sicherheit (Security Modell von Java).
- ActiveX Controls können auch in Java geschrieben werden. Allerdings muss dazu eine COM Implementation mit passenden Interfaces zur verfügung stehen. In der Regel werden ActiveX Anwendungen in Visual Basic oder C++ geschrieben und als DLLs abgespeichert.
- Es steht eine Java-ActiveX Bridge zur Verfügung, die gratis von Sun heruntergeladen werden kann:
  http://java.sun.com/products/javabeans/software/

  Zudem steht auch eine Migrationssoftware ab der gleichen Adresse (ActiveX zu JavaBeans) zur Verfügung.

In der Literatur finden Sie verschiedene Vergleiche. Hier zwei Links zu Artikeln in JavaWorld:

1) http://www.jawaworld.com/javaworld/jw-02-1997/jw-02-activex-beans.html
2) http://www.jawaworld.com/javaworld/jw-03-1997/jw-03-avb-tech.html

1.2.2.6. JavaBeans Benefit Analyse

Write Once, Run Anywhere™	`java.beans` Package ist Teil des Core API
Component Reusability	Reuse Everywhere - plattformunabhängig Beispiele: 3D Charting Bean
Interoperability	Kommunikation mit anderen Komponenten-Architekturen Beans-ActiveX Bridge, Beans-OpenDoc

3) http://www.jawaworld.com/javaworld/jw-02-1997/jw-02-activex-beans.html

1.2.3. Schreiben von Beans Komponenten

Bevor Sie eine Bean kreieren, müssen Sie festlegen, was diese tun soll: Events, properties und Methoden. Die meisten Methoden ergeben sich aus der Definition der Events und Properties der bean.

1.2.3.1. Events

Ein Event gestattet es Ihrer Bean zu kommunizieren, sofern irgend etwas Interessantes geschieht. Kommunikation besteht aus drei Teilen:

- EventObject - dem Ereignisobjekt (dem Ereignis als Objekt dargestellt)
- EventListener - (dem Ziel der Kommunikation)
- Event Quelle (die Bean)

1.2.3.1.1. EventObject

Die `java.util.EventObject` Klasse ist die Basis für Beans Events.

```
public class java.util.EventObject
    extends Object implements java.io.Serializable {
  public java.util.EventObject (Object source);
  public Object getSource();
  public String toString();
}
```

Obschon Sie `EventObject` Instanzen direkt kreieren könnten, geben Ihnen die Design Patterns den Rat, Unterklassen für die einzelnen spezifischen Event-Typen zu kreieren.

Hier ein Beispiel:

```
public class HireEvent extends EventObject {
  private long hireDate;
  public HireEvent (Object source) {
    super (source);
    hireDate = System.currentTimeMillis();
  }
  public HireEvent (Object source, long hired) {
    super (source);
    hireDate = hired;
  }
  public long getHireDate () {
    return hireDate;
  }
}
```

1.2.3.1.2. EventListener

Das `EventListener` Interface ist leer. Aber es dient als Basis für alle Event Listener. Diese müssen dieses Interface implementieren. Ein Listener benötigt eine Nachricht und einen Sender. Der Listener empfängt das spezifische EventObject bzw. deren Unterklasse als einen Parameter. Die Namensgebung ist selbstsprechend: zum `HireEvent` gehört der `HireListener:`

```
public interface HireListener
    extends java.util.EventListener {
  public abstract void hired (HireEvent e);
}
```

1.2.3.1.3. Event Source - der Event Verursacher

Ohne Event Source wären das Event Objekt und der Listener mehr oder weniger nutzlos. Die Event Source definiert wann und wo ein Ereignis geschieht. Klassen, welche sich für ein Ereignis interessieren, müssen sich selbst registrieren und erhalten dann eine Nachricht, wenn etwas (das Ereignis) passiert. Das Registrieren bzw. Entfernen aus der Liste geschieht mit folgenden Methoden

```
public synchronized void add<ListenerType>(<ListenerType> l);
public synchronized void remove<ListenerType>(
  <ListenerType> l);
```

Die Event Quelle muss die Liste selber unterhalten:

```
private Vector hireListeners = new Vector();
public synchronized void addHireListener (
  HireListener l) {
  hireListeners.addElement (l);
}
public synchronized void removeHireListener (HireListener l) {
  hireListeners.removeElement (l);
}
```

Falls Sie AWT Events verwenden, stellt Ihnen AWT diese Infrastruktur bereits zur Verfügung. In diesem Fall müssen Sie die Liste der Listener nicht mehr selber aufbauen und unterhalten.

Falls Sie lediglich einen einzelnen Listener zulassen, können Sie entweder eine Variante des Singleton Patterns einsetzen oder einfach die Exception `java.util.TooManyListenerException` einbeziehen. Diese Exception wird dann jeweils geworfen, falls mehrere Listener registriert werden sollen. Natürlich wird zur Verwaltung einer solchen 'Liste' auch kein Vektor mehr benötigt. Wichtig wird aber die Berücksichtigung der Concurrency, der Nebenläufigkeit, da sonst Race Conditions auftreten könnten.

Die benachrichtigung der Listener könnte beispielsweise folgendermassen aussehen:

```
protected void notifyHired () {
        Vector l;
        // kreiere Event
        HireEvent h = new HireEvent (this);
        // Sicherheitskopie des Vektors
        synchronized (this) {
                l = (Vector)hireListeners.clone();
        }
        for (int i=0;i<l.size();i++) {
                HireListener hl = (HireListener)l.elementAt (i);
                hl.hired(h);
        }
}
```

1.2.3.2. Properties

Properties sind öffentliche Attribute für Beans. Diese werden in der Regel durch nicht öffentliche Instanzvariablen repräsentiert, als read-write, read-only oder write-only.

Generell unterscheidet man folgende Typen von Properties:

- simple
- indexierte
- gebundene / bindende
- eingeschränkte / einschränkende

Die Properties werden wir im praktischen Teil konkret im Einsatz sehen. Dort dürfte Ihnen die folgende Theorie auch besser verständlich werden, falls Sie Mühe haben, im Detail zu folgen.

1.2.3.2.1. Simple Properties

Wie der Name bereits sagt, handelt es sich bei diesen Properties um die einfachst möglichen. Um solche Parameter zu definieren, muss man einfach ein Paar Methoden (set/get) definieren. Der Name der Methode entspricht dabei dem Namen der Property. Der Property Namen entspricht in der Regel gleich dem Variablennamen.

Hier ein Beispiel:

Name der Property : `gehalt` (einer Mitarbeiter-Bean)

```
float salary;
public void setGehalt(float neuesGehalt) {
        gehalt = neuesGehalt;
}
public float getGehalt() {
        return gehalt;
}
```

Falls Sie eine read-only Property wollen, definieren Sie keine set Methode; falls Sie eine write-only Property möchten, können Sie die get Methode weglassen.

Falls Ihre Eigenschaft / Property Boole'sche Werte annimmt, können Sie anstelle der set Methode auch eine `isPropertyName()` Methode definieren:

```
boolean trained;
public void setTrained (boolean trained) {
        this.trained = trained;
}
public boolean isTrained () {
        return trained;
}
```

1.2.3.2.2. Indexierte Properties

Sie benötigen indexierte properties, falls ein Property mehrere Werte aufnehmen kann, also ein Datenfeld, ein Array. Und so sieht das Pattern in diesem Fall aus:

```
public void setPropertyName (PropertyType[] list)
public void setPropertyName (PropertyType element, int position)
public PropertyType[] getPropertyName ()
public PropertyType getPropertyName (int position)
```

Und hier ein Beispiel aus AWT:

```
public class ListBean extends List {
        public String[] getItem() {
                return getItems();
        }
        public synchronized void setItem(String item[]) {
                removeAll();
                for (int i=0;i<item.length;i++)
                        addItem (item[i]);
        }
        public void setItem(String item, int position) {
                replaceItem(item, position)
        }
}
```

Die Methode `String getItem(int pos)` ist bereits in der Klasse `List` definiert.

1.2.3.2.3. Gebundene / Bound Properties

Das folgende Beispiel zeigt eine mögliche Anwendung gebundener Properties. Dabei geht es um Folgendes:

ein Objekt möchte bei Änderungen, die ein anderes Objekt betreffen, informiert werden.
Beispiel:
Sie möchten informiert werden, wenn Ihr Chef einen Bonus >= 200'000 CHF erhält.

Damit eine solche Verbindung funktioniert, müssen Sie eine Watch Liste für die `PropertyChangeEvent` s der `PropertyChangeSupport` Klasse unterhalten:

1) kreieren Sie eine Liste der Listener:
```
private PropertyChangeSupport changes = new PropertyChangeSupport
(this);
```

2) unterhalten Sie diese Liste
```
public void addPropertyChangeListener (
   PropertyChangeListener p) {
   changes.addPropertyChangeListener (p);
}

public void removePropertyChangeListener (
   PropertyChangeListener p) {
         changes.removePropertyChangeListener (p);
   }
}
```
3) die Änderung des Bonus muss mit dem Event verknüpft werden

```
public void setGehalt (float gehalt) {
   Float altesGehalt = new Float (this.gehalt);
   this.gehalt = Gehalt;
   changes.firePropertyChange("Gehalt",altesGehalt,new
                                    Float(this.gehalt));
}
```

4) am empfangenden Ende könnten Sie beispielsweise eine `change...()` Methode definieren:
```
public void propertyChange(PropertyChangeEvent e);
```

Java informiert die Klasse / Beans über `PropertyChangeEvent` s.

1.2.3.2.4. Constrained Properties

Constrained Properties verhalten sich analog zu den Bound Properties. Allerdings wird neben der Liste der `PropertyChangeListener` eine Liste der `VetoableChangeListener` unterhalten. Bevor ein bean eine Änderung aktuell durchführt, überprüft es die Liste der Veto Listener.Falls die Änderung nicht gestattet wird, kann der Listener eine `PropertyVetoException` werfen. Diese wird mit der `set...()` Methode definiert.

Als Beispiel können wir wieder unser Gehaltsereignis hernehmen.Als erstes müssen wir die Klasse ergänzen:

```
private VetoableChangeSupport vetoes = new VetoableChangeSupport (this);
public void addVetoableChangeListener (VetoableChangeListener v) {
        vetoes.addVetoableChangeListener (v);
}
public void removeVetoableChangeListener (VetoableChangeListener v) {
        vetoes.removeVetoableChangeListener (v);
}
```

und anpassen:

```
public void setGehalt(float gehalt) throws PropertyVetoException {
        Float altesGehalt = new Float (this.gehalt);
        vetoes.fireVetoableChange ("Gehalt", altesGehalt,
                                                new Float(gehalt));
        this.gehalt = gehalt;
        changes.firePropertyChange ("Gehalt", altesGehalt,
                                          new Float(this.gehalt));
}
```

Der Empfänger, der `VetoableChangeListener`, benötigt eine `vetoableChange()` Methode:

```
public void vetoableChange(PropertyChangeEvent e)
                                       throws PropertyVetoException;
```

Anstatt je einen Listener für die Property Änderungen und einen für die Veto gesteuerten Events zu unterhalten, kann man separate Listen für jede Eigenschaft unterhalten. Dies führt zu folgendem 'Design Pattern':

```
public void add<i>PropertyName</i>Listener(PropertyChangeListener p);
public void remove<i>PropertyName</i>Listener(PropertyChangeListener p);
```

und

```
public void add<i>PropertyName</i>Listener(VetoableChangeListener v);
public void remove<i>PropertyName</i>Listener(VetoableChangeListener v);
```

1.2.3.3. Methoden

Methoden gestatten es uns ein Bean sinnvoll zu nutzen. Dabei kann irgend ein programm die Methoden eines Bean aufrufen, sofern diese öffentlich ist. Sie werden in der Regel Methoden für die Ereignisse und Properties kreieren. Aber zusätzlich sind auch Methoden nötig, welche unterstützende Aufgaben übernehmen.

In den Beispielen werden Sie beliebig viele Methoden selber definieren und modifizieren können und müssen. Neues aus Java Sicht gibt es nichts!

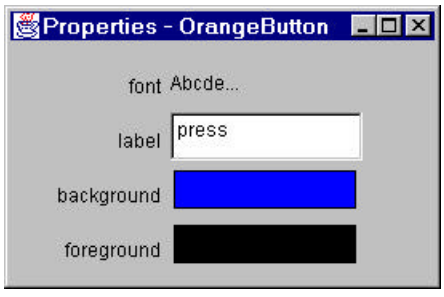
1.2.3.4. Customization

Mit der Customization können Sie als bean Entwickler den Zugriff des Bean Integrators auf Ihre Beans kontrollieren, was er in seinem Builder Tool sehen und einsetzen kann. Builder Tools verwenden das Refelction Interface, um festzustellen, welche Attribute und Methoden eine JavaBean zur Verfügung stellen kann.

Sie haben aber zusätzlich die Möglichkeit ein `Customizer` Interface zu implementieren und ein `Panel` zur Verfügung zu stellen. Im BDK können Sie den Customizer als separates Fenster sehen, sofern Sie ihn unter *Edit* in der BeanBox auswählen. Nicht alle beans verfügen über einen Customizer. Standardmäsig werden mehrere Beans mit dem BDK mitgeliefert. Wenn Sie beispielsweise die JDBC Bean in die BeanBox plazieren, dann können Sie deren Customizer aktivieren. Hier ein Beispiel eines Buttons mit Customizer aus den BDK Beispielen:

links sehen Sie die beanBox, oben das Customizer Fenster Im Falle der JDBC Bean können Sie die URL der DB sowie benutzername und Passwort angeben.

Daneben können Sie auch die Klasse `PropertyEditorSupport` erweitern. In diesem Fall sehen Sie im Property Fenster folgende Angaben:



Dabei habe ich durch Doppelklick auf das Background Feld bei der Farbauswahl 'orange' durch 'blau' ersetzt.

Im BDK 1.1 wird das Property Fenster automatisch aufgemacht und ist immer offen.

Sie finden eine Liste der BDK Beispiele in der HTML Datei examples.html im Verzeichnis doc. Dort stehen auch die Angaben, ob Customizer oder was auch immer aktiviert / implementiert ist.

Diese beiden Mechanismen bedingen, dass Sie zusätzlich zur Bean Klasse auch noch eine unterstützende Klasse definieren und implementieren. Dies zweite Klasse implementiert das `BeanInfo` Interface

1.2.3.5. BeanInfo

Das BeanInfo Interface gestattet es Ihnen Ihre Beans detaillierter zu beschreiben als Reflection selbständig bestimmen kann. In der Regel verwenden Sie eine `BeanInfo` implementierende Klasse, um Ihre Bohne handlicher zu machen, selbsterklärend und um dem Benutzer eventuell weniger Optionen zur Verfügung zu stellen.

Damit iese Klasse auch eingesetzt werden kann, muss sie so benannt werden wie die Bean, plus dem Zusatz `BeanInfo` am Ende.

Beispiel:

Name der JavaBean: `MeineBohne`
Name der BeanInfo: `MeineBohneBeanInfo`

Java stellt Ihnen zusätzlich die SimpleBeanInfo Klasse. Diese nimmt Ihnen bereits einiges ab. Sie können damit schneller einfache BeanInfo Klassen kreieren, indem Sie selektiv überschreiben.

1.2.3.6. Übungen

Wenn Sie nun eine Theoriepause machen möchten und sich die Konzepte in der Praxis anschauen möchten, können Sie folgende Übungen durcharbeiten:

Sie finden diese Übungen mit Lösungshinweisen und einer Musterlösung und dem erwarteten Verhalten (als Screen Shots) im zu dieser Kurseinheit gehörenden Übungsteil.

1) Events und Properties
2) Reflection / Introspection und BeanInfo
3) BeanInfo mit Zusatzinformationen
4) Security (X.509) und JavaBeans (Übung)

1.2.4. Design-time vs. Run-time Beans "mode"

Beans müssen sowohl innerhalb eines Laufzeitsystems als auch innerhalb des Builders funktionieren. Zur Designzeit muss die Bean Informationen liefern und akzeptieren, welche benötigt wird, um Eigenschaften zu verändern oder zu definieren (siehe oben: Farbe, Text, ...). Zusätzlich müssen auch die Methoden und Ereignisse sichtbar oder bekannt sein.

Zur Laufzeit ist ein anderes Verhalten gefragt. In diesem Mode soll die Bean primär ihren Dienst verrichten, beispielsweise Daten grafisch aufbereiten oder emails abfragen oder .... Dieses Verhalten unterscheidet sich wesentlich vom Entwurfszeitverhalten.

1.2.5. Introspection

Introspection ist ein Prozess, bei dem untersucht wird, welche Methoden, Events und Properties durch ein JavaBean unterstützt werden. Dazu stellt das JavaBean Package eine spezielle Klasse, die `Introspection` Klasse zur Verfügung. Diese basiert auf dem Reflection API, welches Sie aus Java 2 kennen. Die Bean Informationen aus BeanInfo werden mit einer Methode `getBeanInfo()` bestimmt.

```
TextField tf = new TextField ();
BeanInfo bi = Introspector.getBeanInfo (tf.getClass());
```

Falls Sie BeanInfo nicht definiert haben, wird einfach das Reflection API an Stelle von BeanInfo verwendet.

Im Folgenden schauen wir uns an, was Introspection genau liefert, getrennt für die drei Bereiche:

- Events,
- Properties und
- Methoden.

1.2.5.1. Events

Die Methode `getEventSetDescriptors()` Methode berichtet über alle Events, welche diese Bean feuern kann. Pro Set add/removeListenerTyp Methode wird ein Event Set für die Bean definiert.

```
EventSetDescriptor[] esd = bi.getEventSetDescriptors();
for (int i=0;i<esd.length;i++)
                System.out.print (esd[i].getName() + " ");
System.out.println ();
```

Beispiel:
Im Falle eines Text Feldes würde die obige Zeile zu folgender Ausgabe führen

```
text mouse key component action focus mouseMotion
```

1.2.5.2. Properties

Die getPropertyDescriptors() Methode liefert alle Properties einer Bean. Ein Property wird durch eine oder mehrere Methoden gemäss dem folgenden Muster definiert:

```
public void set PropertyName (PropertyType value);
public PropertyType get PropertyName ();
public boolean is PropertyName ();
PropertyDescriptor pd[] = bi.getPropertyDescriptors();
for (int i=0;i<pd.length;i++)
        System.out.print (pd[i].getName() + " ");
System.out.println ();
```

Beispiel:
Im Falle eines Text Feldes führt dies zu folgender Ausgabe

```
selectionStart enabled text preferredSize
foreground visible background selectedText
echoCharacter font columns echoChar name
caretPosition selectionEnd minimumSize editable
```

1.2.5.3. Methoden

Die `getMethodeDescriptors()` Methode liefert alle Methoden der Bean. Die Methode liefert einfach eine Liste aller öffentlichen Methoden. Damit haben Sie die Möglichkeit eine dieser Methoden aufzurufen, ohne dass Sie diese im voraus bereits kennen müssen.

Die Parameter der Methoden erhalten Sie mit der `getParameterDescriptors()` Methode.

Muster:

```
MethodDescriptor md[] = bi.getMethodDescriptors();
for (int i=0;i<md.length;i++)
        System.out.print (md[i].getName() + " ");
System.out.println ();
```

Beispiel:
Im Falle des Text Feldes erhalten Sie eine Liste von mehr als 150 Methoden, inklusive jener der Oberklassen und der add/remove Event Listener Methoden.

1.2.5.4. BeanInfo

Bisher haben wir in unseren Beispielen angenommen, dass `Introspection` mit `BeanInfo` zusammenarbeitet. Falls Sie aus irgend einem Grund keine BeanInfos verwenden, wird das Refelction API eingesetzt, um Methoden, Events und Properties zu bestimmen.

Falls Sie wollen, dass nicht alle Informationen zur Verfügung stehen, können Sie die entsprechenden Methoden in einer eigenen `BeanInfo` überschreiben.

Beispiel:
wir betrachten wieder ein Text Feld und definieren eine passende `BeanInfo`.
Dann könnte unsere `BeanInfo` schematisch folgendermassen aussehen:

```
import java.beans.*;
public class SizedTextFieldBeanInfo extends SimpleBeanInfo {
        private final static Class beanClass =
                                        SizedTextField.class;
        public PropertyDescriptor[] getPropertyDescriptors() {
                try {
                        PropertyDescriptor length =
                        new PropertyDescriptor("length", beanClass);
                        PropertyDescriptor rv[] = {length};
                        return rv;
                } catch (IntrospectionException e) {
                        throw new Error(e.toString());
                }
        }
}
```

Nun werden Sie nicht mehr die 10+ Eigenschaften des Text Feldes erhalten, sondern nur noch die Länge! Sie könnten dann auch noch spezielle get/set Methoden definieren, um das System perfekt zu machen.

Sie können analog vorgehen, wenn Sie beispielsweise die Ausgabe formattieren möchten. Das folgende Beispiel zeigt Ihnen wie so etwas aussehen könnte:

```
public BeanDescriptor getBeanDescriptor() {
        BeanDescriptor bd = new BeanDescriptor(beanClass);
        bd.setDisplayName("Sized Text Field");
        return bd;
}
```

Wenn Sie noch einen Schritt weiter gehen wollen, können Sie auch noch ein Icon zur Verfügung stellen:

```
public Image getIcon (int iconKind) {
        if (iconKind == BeanInfo.ICON_COLOR_16x16) {
                Image img = loadImage("sized.gif");
                return img;
        }
        return null;
}
```

1.2.6. Customizers

Customizer gestatten es *Ihnen* zu definieren, wie der Benutzer Ihrer beans diese konfigurieren kann. Sie haben mehrere Möglichkeiten Ihren eigenen Customizer zu schreiben. Wie dies aussehen könnte, sehen Sie im folgenden Abschnitt

1.2.6.1. Schreiben eigener Customizers

Nehmen wir an, Ihre Bohne besitze lediglich eine Gehalts-Property. In diesem Fall könnten Sie einen Customizer definieren, der lediglich die Eingabe nummerischer Zeichen im Eingabefeld gestattet. Customizers sind allgemein gesprochen viel mächtiger. Aber zur Illustration des Konzepts ist es besser das Beispiel einfach zu halten.

```
package mitarbeiter;

import java.awt.*;
import java.awt.event.*;
import java.beans.*;

public class MitarbeiterCustomizer extends Panel
                                implements Customizer, KeyListener {
        private Mitarbeiter target;
        private TextField gehaltsFeld;
        private PropertyChangeSupport support =
                                        new PropertyChangeSupport(this);
        public void setObject(Object obj) {
                target = (Mitarbeiter) obj;
                Label t1 = new Label("Gehalt :");
                add(t1);
                gehaltsFeld = new TextField(
                String.valueOf(target.getGehalt()), 20);
                add(gehaltsFeld);
                gehaltsFeld.addKeyListener(this);
        }
        public Dimension getPreferredSize() {
                return new Dimension(225,50);
        }
        public void keyPressed(KeyEvent e) {}
        public void keyTyped(KeyEvent e) {}
```

```
        public void keyReleased(KeyEvent e) {
                Object source = e.getSource();
                if (source==gehaltsFeld) {
                        String txt = gehaltsFeld.getText();
                        try {
                                target.setGehalt(
                                        (new Float(txt)).floatValue());
                        } catch (NumberFormatException ex) {
                                gehaltsFeld.setText(
                                String.valueOf(target.getGehalt()));
                        }
                        support.firePropertyChange("", null, null);
                }
        }
        public void addPropertyChangeListener(
                PropertyChangeListener l) {
                        support.addPropertyChangeListener(l);
        }
        public void removePropertyChangeListener(
                PropertyChangeListener l) {
                support.removePropertyChangeListener(l);
        }
}
```

wobei wir auch noch folgende `BeanInfo` Klasse definiert haben:

```
package mitarbeiter;
import java.beans.*;
public class MitarbeiterBeanInfo extends SimpleBeanInfo {
        public BeanDescriptor getBeanDescriptor() {
                return new BeanDescriptor(
                        beanClass, customizerClass);
        }
        private final static Class beanClass =
                Mitarbeiter.class;
        private final static Class customizerClass =
                MitarbeiterCustomizer.class;
}
```

1.2.6.2. Schreiben eines Property Customizers

Wie oben erwähnt, kann man auch einen Property Customizer definieren und jeweils anzeigen lassen. Sie müssen dazu einen `PropertyEditor` implementieren. Sie können dabei entweder die Klasse selber implementieren oder analog zu den BeanInfos mit einer vorbereiteten Klasse starten: `PropertyEditorSupport`.

Beispiel:
wir könnten die Stellung des Mitarbeiters in der Firma als Property definieren und die möglichen Positionen auflisten.

```
import java.beans.*;
public class EmployeePositionEditor extends PropertyEditorSupport {
        public String[] getTags() {
                String values[] = {
                        "CEO",
                        "COO",
                        "CTO",
                        "CIO"};
                return values;
        }
}
```

Nun können wir die beanInfo Klasse passend ergänzen:

```
public PropertyDescriptor[] getPropertyDescriptors() {
        try {
                PropertyDescriptor pd =
                        new PropertyDescriptor("position", beanClass);
                pd.setPropertyEditorClass(positionEditorClass);
                PropertyDescriptor result[] = { pd };
                return result;
        } catch (Exception e) {
                System.err.println("Unexpected exception: " + e);
                return null;
        }
}
```

1.2.6.3. Einsatz des System Property Editors

JDK liefert Ihnen bereits einige passende Property Editoren, die Sie einfach einsetzen können. Auch kommerzielle Tools stellen Ihnen einiges an Hilfswerkzeugen zur Verfügung.

Datatype	**Editor**
Boolean	BoolEditor
Byte	ByteEditor
Color	ColorEditor
Double	DoubleEditor
Float	FloatEditor
Font	FontEditor
Int	IntEditor
Long	LongEditor
Number	NumberEditor
Short	ShortEditor
String	StringEditor

Die obigen Editoren sind im Package `sun.beans.editors` enthalten, also nicht Bestandteil von `java.*` oder `javax.*`. Aber das BDK liefert Ihnen genügend Hilfsklassen und Editoren. Im Praxisteil (im Anhang) finden Sie Hinweise, was Ihnen im JBuilder zur Verfügung steht.

1.2.7. Persistenz

Unter Persistenz versteht man die dauerhafte Abspeicherung des Objektzustandes, dauerhaft im Gegensatz zu temporär, also beispielsweise lediglich während der aktiven Ausführung der Programme.

JavaBeans benutzen dazu die Objektserialisierung. Im einfachsten Fall wird dazu einfach das `Serializable` Interface implementiert. Das Interface besitzt keine Methoden, es muss also nichts Spezielles implementiert werden.

1.2.7.1. Bean Serialisierung

Die Serialisierung eines Objekts geschieht mit der `ObjectOutput.writeObject()` Methode. Als Parameter wird das Objekt angegeben. Zum Deserialisieren wird die Methode `ObjectInput.readObject()` verwendet. Als Bean Entwickler haben Sie jedoch damit nichts zu tun! Ihre Aufgabe ist es zu überprüfen, ob Ihr Objekt auch passend serialisiert werden kann:

1) ist Ihre Klasse serialisierbar?
2) sind alle Instanzenvariablen serialisierbar?
3) wollen Sie alles, was nicht statisch oder transient ist, serialisieren?
4) soll die gesamte Objektstruktur serialisiert werden?
5) wie sollen transiente und statische Variablen behandelt werden, speziell bei der Deserialisierung?
6) werden zusätzliche Validierungen benötigt?

Falls Sie bessere Kontrolle über den Serialisierungsprozess benötigen, können Sie auch auf die Externalisierung ausweichen.

Hier ein Beispiel: (auf dem Server / der CD)

```
public class TreeNode implements Serializable {
  Vector children;
  TreeNode parent;
  String name;
  transient Date date;
  public TreeNode(String s) {
    children = new Vector(5);
    name = s;
    initClass();
  }
  private void initClass () {
    date = new Date();
  }
  ...
  private void writeObject(ObjectOutputStream s)
      throws IOException {
    s.defaultWriteObject();
  }
  private void readObject(ObjectInputStream s)
      throws ClassNotFoundException, IOException {
    s.defaultReadObject();
    initClass();
  }
}
```

1.2.7.2. Bean Rekonstitution

Im Gegensatz zu normalen Objekten werden Beans nicht mit dem `new` Operator, sondern mit der `instantiate()` Methode kreiert.

Mit dieser Technik können Sie auch ein Text Feld kreieren:

```
Component c =
  (Component)Beans.instantiate(
  null, "java.awt.TextField");
```

Dieses Konstrukt ist äquivalent zu

```
public class MyTextField extends java.awt.TextField {
}
```

Un hier noch ein komplexeres Beispiel:

```
import java.awt.*;
import java.io.*;

public class TfSaver {
  public static void main(String args[]) {
    MyTextField mtf = new MyTextField();

    // set the properties
    Font ft = new Font("Serif", Font.ITALIC, 36);
    mtf.setFont(ft);
    mtf.setText("Hello World");

    // serialize
    try {
      FileOutputStream f = new FileOutputStream(
        "MyTextField.ser");
      ObjectOutputStream s =
        new ObjectOutputStream(f);
      s.writeObject(mtf);
      s.flush();
    } catch (Exception e) {
      System.out.println(e);
    }
    System.exit(0);
   }
}
```

Mit der folgenden Anweisung können Sie ein Text Feld kreieren, welches die obigen Eigenschaften besitzt, beispielsweise 36 Punkt Font und den Text "Hello World":

```
Component c =
  (Component)Beans.instantiate(null, "MyTextField");
```

Falls ein serialisiertes Objekt vorhanden wäre, `MyTextField.ser`, kann diese deserialisiert werden; sonst wird eine neue Instanz kreiert.

1.2.7.3. Bean Versioning

Änderungen an Beans führen zu Veränderungen der persistenten Struktur. Mit dem Serialisierungs- / Deserialisierungs- Mechanismus muss man aufpassen: sobald Sie irgend eine Änderung an der Klasse vornehmen, können Sie die serialisierten Objekte nicht mehr weiterverwenden. Solche Änderungen führen zu einer `InvalidClassEXception`. Auf der anderen Seite kann ja eine Änderung zum selben serialisierten Objekt führen. Es müsste also eine Variante geben, die es einem gestattet, auch bei Änderungen immer noch auf serialisierte Objekte zuzugreifen.

Die Serialisierung wird über einen Stream Unique Intentifier SUID kontrolliert.

```
private static final long serialVersionUID =
    -2966288784432217853L;
```

Falls dieser angepasst wird, können Sie auch auf die alten serialisierten Objekte zugreifen bzw. die Unterschiede feststellen und verwalten.

1.2.7.4. Übungen

Jetzt müssen Sie sich intensiv um die Praxis kümmern!
Der Praxisteil liefert Ihnen dazu reichlich Gelegenheit, auch mit dem JBuilder.

1.2.8. Resourcen

Die Standard Informationsquelle für Beans ist der Java Web Site von Sun:

http://java.sun.com/products/javabeans/

The JavaBeans Specification and Tutorial (Addison-Wesley)

Developing Java Beans (O'Reilly)

Design Patterns - Elements of Reusable Object-Oriented Software (Addison-Wesley) Gamma, Helm, Johnson, Vlissides (ISBN 0-201-63361-2).

JAVABEANS ÜBERSICHT