JavaBeans 1.0.1

Josef Templ

14. Oktober 2002

Übersicht

- Standard zum Konfigurieren und Verbinden von Objekten
- Events, Unbound/Bound/Constraint-Properties
- Packaging und Manifest
- BeanInfo
- PropertyEditor, Customizer
- Serialisierung
- BeanContext, JAF

JavaBean Referenzen

- `http://java.sun.com/products/javabeans/`
  JavaBean Home Page

- `http://java.sun.com/docs/books/tutorial/javabeans/`
  JavaBean Tutorial

- `http://java.sun.com/products/javabeans/training.html`
  Ein weiteres JavaBean Tutorial

Definition und Abgrenzung

Definition: *A JavaBean is a reusable software component that can be manipulated visually in a builder tool.*

JavaBeans können sichtbar sein (AWT-Komponenten) oder unsichtbar.

In jedem Fall können sie visuell bearbeitet werden.

Abgrenzung: JavaBeans ist kein Dokumentenstandard wie etwa OLE oder OpenDoc, vergleichbar eher mit ActiveX.

JavaBeans sind entweder durch Klassen oder durch serialisierte Prototypen repräsentiert.

Paket ist `java.beans` und Unterpaket(e).

Anforderungen an eine JavaBean-Klasse

Parameterloser öffentlicher Konstruktor: damit Builder-Tool eine Instanz erzeugen kann.

Serialisierbar (java.io.Serializable, java.io.Externalizable): damit serialisierte Prototypen erstellt werden können.

Keine spezielle Basisklasse oder Interface für Beans.

Schnittstelle für Builder-Tool:

- Implizit durch Berücksichtigung von Namenskonventionen
- Explizit durch BeanInfo-Klasse

Serialisierung/Deserialisierung

Verwendet `ObjectOutputStream` und `ObjectInputStream`.

Problem bei der Deserialisierung: Zum dynamischen Laden von Klassen wird ein ClassLoader verwendet, welcher? Kann bei `ObjectInputStream` nicht angegeben werden.

Abhilfe:
`java.beans.Beans.instantiate(ClassLoader cl, String beanName)`

Wobei `beanName` in Punktnotation ohne Extension `.class` oder `.ser` ist, z.B. `my.package2.MyBean2`.

Erweiterungen für Speicherung im XML-Format.

JavaBean Bestandteile

- Events
- Properties
- Methoden

Events

Events sind Ereignisse, die von einem Bean an registrierte Objekte (Listeners) geschickt werden.

Beispiel: WindowEvent, MouseEvent, PropertyChangedEvent, etc.

Verwaltung der Listeners durch Bean-Klasse.

Normalerweise Multicast ohne definierte Reihenfolge.

Unicast möglich.

Registrierung und Entfernung von Listeners durch Standard-Pattern.

Vergleich mit .NET: Konzeptuell gleich aber syntaktische Unterstützung für Listener-Verwaltung und Aufruf durch Modifier `event` und Typ `delegate`.

Pattern für Ereignis Xxx

```
class XxxEvent extends java.util.EventObject {...}
interface XxxListener extends java.util.EventListener {...}
class MyBean {...
    public void addXxxListener(XxxListener l) {...}
    public void removeXxxListener(XxxListener l) {...}
}
class MyClass implements XxxListener {...}

MyBean mb ...
mb.addXxxListener(new XxxListener() { //anonymous inner class
    ...
}
```

XxxListener

Enthält Methode(n) zur Event-Behandlung.

Mindestens eine Methode erforderlich, aber mehrere möglich, da sonst zu viele Listeners.

Beispiel *WindowEvent* mit *windowClosed*, *windowOpend*, *windowActivated*, ...

Empfehlung: Jede Methode hat genau einen Parameter vom Typ XxxEvent.

Achtung: nicht jedes Builder-Tool unterstützt allgemeine Parameterlisten.

Bei Listeners mit mehreren Methoden häufig Hilfsklasse (XxxAdapter) mit leerer Implementierung für selektives Überschreiben einzelner Methoden.

Unicast und Multicast

Im Normalfall Multicast mit beliebiger Anzahl von Listeners.

In Ausnahmefällen Unicast.

Unicast-Listener gekennzeichnet in addXxxListener durch

```
throws java.util.TooManyListenersException
```

Unicast ist Spezialfall von Multicast, daher Upgrade möglich ohne Code zu invalidieren.

Properties

JavaBeans können eine beliebige Anzahl sogenannter *Properties* besitzen.

Ein Property kann über das Builder-Tool gelesen und (optional) geschrieben werden.

Für das Lesen und Schreiben von Properties werden Methoden (*getter* und *setter*) verwendet.

Der Propertytyp drückt sich in der Methodensignatur aus (z.B. String).

Die Menge der Properties und deren Zugriffsmethoden wird für eine Klasse `X` über eine Hilfsklasse mit dem Namen `XBeanInfo` definiert.

Property Pattern

Wenn die BeanInfo-Klasse fehlt, wird Introspection verwendet.

Allgemeines Pattern für Property `X` vom Typ `T`:

```
public void setX(T val);
public T getX();
```

Pattern für Property `X` vom Typ `boolean`:

```
public void setX(boolean val);
public boolean isX();
```

Indexed Properties

Pattern für Property `X` vom Typ `T[]`.

Enthält zusätzliche Methoden für elementweisen Zugriff.

```
public void setX(T[] val);
public void setX(ind index, T val);
public T[] getX();
public T getX(int index);
```

Bound/Constrained Properties

Sind an `java.beans.PropertyChangeEvent` gebunden.

Rufen registrierte `java.beans.PropertyChangeListener` auf, wenn geändert.

Bound Properties erfordern keine eigenen Events und Listeners.

Unterstützt durch Klasse `java.beans.PropertyChangeSupport`.

Constrained Properties erlauben im Listener das Werfen einer `java.beans.PropertyVetoException`.

Dadurch wird das Setzen des neuen Wertes unterbunden.

Constrained Properties nur in Ausnahmefällen verwenden!

BeanInfo

Erlaubt Internationalisierung von JavaBeans.

Erlaubt Abweichung von und Mischung mit den Standardpatterns.

Erlaubt beliebige Einschränkung der publizierten Schnittstelle eines Beans.

Stellt Descriptor-Objekte zur Verfügung für: gesamtes Bean, Property, etc.

Jeder Descriptor ist von `java.beans.FeatureDescriptor` abgeleitet.

Ein FeatureDescriptor enthält immer Namen, Beschreibung, ...

Klasse `SimpleBeanInfo` kann als Basisklasse verwendet werden.

SimpleBeanInfo

```
class XxxBeanInfo extends SimpleBeanInfo {
  // noops, defaults to introspection
}
```

Selektives Überschreiben einzelner Methoden möglich, z.B.:

```
public Image getIcon(int iconKind)
  return loadImage("MyBean16.gif");
}
public BeanDescriptor getBeanDescriptor()
  return new BeanDescriptor(MyBean.class,
                            MyBeanCustomizer.class);
}
```

FeatureDescriptor

Basisklasse für BeanDescriptor, EventSetDescriptor, MethodDescriptor, ParameterDescriptor, PropertyDescriptor.

Jeder FeatureDescriptor enthält:

- Internen und angezeigten Namen des Features
- Expert, Preferred, Visibility
- Kurzbeschreibung
- Lesen und Setzen des Wertes

BeanDescriptor

von BeanInfo geliefert.

Beschreibt globale Bean-Eigenschaften.

```
public class BeanDescriptor extends FeatureDescriptor {
  public BeanDescriptor(Class beanClass){...};
  public BeanDescriptor(Class beanClass,
                        Class customizerClass){...};
  public Class getBeanClass(){...};
  public Class getCustomizerClass(){...};
}
```

Customizer

Die Customizer-Klasse stellt ein GUI zum Bearbeiten eines Beans bereit.

Erweitert `java.awt.Component`.

Wird von Builder-Tool zum Bearbeiten eines Beans instanziert und dargestellt.

Muss parameterlosen Konstruktor besitzen.

Meist wird kein eigener Customizer verwendet.

Default meist in Tabellenform mit zwei Spalten (Name, Wert) und für jedes Property und Event eine Zeile.

Property-Editor kann ebenfalls angegeben werden.

BeanInfo.getPropertyDescriptors

Builder-Tool benötigt PropertyDescriptor für jedes editierbare Property.

Default sind alle Getters und Setters, die den Namenskonventionen folgen.

Für Internationalisierung, Einschränkung, Umbenennung, etc. muss get-PropertyDescriptors entsprechend implementiert werden.

```
public PropertyDescriptor[] getPropertyDescriptors() {
  return new PropertyDescriptor[] {
    new PropertyDescriptor("Breite", MyBean.class,
      "getWidth", "setWidth");
    new PropertyDescriptor("Hoehe", MyBean.class,
      "getHeight", "setHeight");
  }
}
```

PropertyEditor

Definiert durch `PropertyDescriptor.getPropertyEditorClass`.

Implementiert `PropertyEditor` oder erweitert `PropertyEditorSupport`.

Muss einen parameterlosen Konstruktor aufweisen.

Methode `setValue(T x)` immer unterstützt.

Drei verschiedene Möglichkeiten:

Text based Methoden `setAsText` und `getAsText` implementiert.

Liste Aufzählung aller möglichen Werte.

Custom Editor `paintValue` und/oder `getCustomEditor` implementiert.

Packaging und Manifest

Ein oder mehrere Beans werden zu einem jar-File verpackt.

Enthält alle Klassen und Ressourcen für Bean(s).

Jar-Manifest (META-INF/MANIFEST.MF) enthält Information über Inhalt.

Folge von Sektionen (durch Leerzeilen gtrennt).

Sektion enthält Information zu einem File.

Information ist zum Beispiel, ob es sich um ein JavaBean handelt.

Bean kann als Klasse (`.class`) oder serialisiert (`.ser`) vorliegen.

Beispiel Manifest

```
Manifest-Version: 1.0

Name: my/package1/MyBean1.class
Java-Bean: True

Name: my/package1/MyBean1BeanInfo.class
Design-Time-Only: True

Name: my/package2/MyBean2.ser
Java-Bean: True
Depends-On: my/package1/MyBean1.class
```

Tags

Folgende Tags werden zur Kennzeichnung von Beans und deren Eigenschaften verwendet:

Java-Bean Wert `True` signalisiert, dass es sich bei dem File in der Sektion um ein JavaBean handelt.

Depends-On Listet ALLE Abhängigkeiten auf (flat). Kann mehrfach pro Sektion vorkommen. Wenn es fehlt sind die Abhängigkeiten UNBEKANNT.

Design-Time-Only Wert `True` signalisiert, dass diese Datei nur zur Entwicklungszeit benötigt wird.

Dokumentation

Pro Bean kann im Jar-File ein HTML-File mit Namen `<beanName>.html` vorkommen.

Für Lokalisierung wird pro Locale ein Unterverzeichnis verwendet, also `<locale>/<beanName>.html`.

Empfohlenes Format ist HTML 2.0.

Interne und externe Referenzen (Hyperlinks) sind erlaubt.

Das verwendete BeanBuilder-Tool ist für die Anzeige zuständig.

NetBeans zeigt die Doku nicht an.

BeanContext

Paket `java.beans.beancontext` definiert Interfaces zur hierarchischen Strukturierung von Beans.

BeanContext ist Container

BeanContextChild ist Element eines Contexts.

AWT-Komponente kann BeanContext nicht implementieren wegen Namenskollision zwischen Collection und Component.

Abhilfe mittels `BeanContextProxy` Interface. Erlaubt Delegation zu Hilfsobjekt.

Ein Objekt kann Services für Children bereitstellen durch Implementieren der Schnittstelle `BeanContextServices`.

JavaBeans Activation Framework (JAF)

Java Platform standard extension `javax.activation`.

Ähnlich zu Windows Filtetype-Associations.

Klasse `DataHandler` stellt Implementierung zur Verfügung und wird von Anwendung verwendet.

Interface `DataSource` abstrahiert von Dateizugriff (z.B FileDataSource, URLDataSource).

Interface `CommandMap` abstrahiert von Zuordnungsmechanismus für Kommandos pro MIME-Typ (z.B. View, Edit für .txt aus mailcap).

Klasse `CommandInfo` stellt Info pro Kommando bereit (Name, Klasse).

Einsetzbar z.B. für File-Manager (Windows Explorer).

JAF Beispiel

```
File file = new File(fileName);
DataSource ds = new FileDataSource(file);
DataHandler dh = new DataHandler(ds);
CommandInfo cmdInfo[] = dh.getPreferredCommands();
int i = ... //selected command index
Object cmd = cmdInfo[i].getCommandObject();
if (cmd instanceof java.awt.Component) {
    ... // display component
}
```