




Tehnici avansate de programare

Curs -

Cristian Frăsinaru

`acf@infoiasi.ro`

Facultatea de Informatică
Universitatea "Al. I. Cuza" Iași





JavaBeans



Cuprins



- Specificatiile JavaBeans
- Tipuri de proprietati
- Crearea unui BeanInfo
- Crearea unui PropertyEditor
- Crearea unui Customizer
- Considerații generale



Ce sunt componentele JavaBeans ?

Componentele **JavaBeans** sunt:

- componente de sine stătătoare, reutilizabile, independente de platformă
- pot fi folosite pe orice platformă de programare
- se conformează cu o serie de specificații
- definesc un model pentru crearea de componente software.

Un **container** reprezintă un context în care componentele pot fi grupate și cu care acestea pot interacționa.

Caracteristici



- **Introspecție** - pentru analizarea componentei de catre container
- **Personalizare** - pentru modificarea aspectului si functionalitatii
- **Proprietăți** - pentru individualizare si programare
- **Evenimente** - pentru comunicare
- **Persistență** - salvare / restaurare proprietăți



Exemplu de bean

```
public class MyBean implements java.io.Serializable {  
  
    private String value;  
  
    public String getValue() {  
        return value;  
    }  
  
    public void setValue(String value) {  
        this.value = value;  
    }  
}
```

Tipuri de proprietăți

Proprietatile unei componente JavaBean sunt elementele care definesc caracteristicile acelei componente.

Concret, proprietatile sunt valorile accesate prin intermediul metodelor publice de tip *getter*, *setter*. Pot fi de patru tipuri:

- simple
- legate
- constrânse
- indexate

Proprietăți simple

Sunt proprietati care influenteaza aspectul sau functionalitatea unei componente dar a caror schimbare nu implica nici un efect colateral.

```
public class Component implements Serializable {
    private Font font;
    private boolean enabled;

    public Font getFont() { return font; }
    public void setFont(String font) {
        this.font = font;
    }
    public boolean isEnabled() { return text; }
    public void setEnabled(boolean enabled) {
        this.enabled = enabled;
    }
}
```

Proprietăți legate

Sunt proprietati care, la modificarea lor, pot **informa** obiectele interesate de tip **listener**, asupra modificarii aparute.

O componenta ce contine proprietati legate trebuie sa ofere un mecanism prin care alte obiecte sa se inregistreze ca ascultatori ai acelor proprietati.

Informarea ascultatorilor asupra schimbarii unei proprietati se face **dupa** ce schimbarea a avut loc prin intermediul unui obiect de tip `PropertyChangedEvent`.

Exemplu PropertyChange (1)

```
public class MyBean {
    private String text;
    private PropertyChangeSupport changes=new PropertyChangeSupport(this);

    public void addPropertyChangeListener(PropertyChangeListener l) {
        changes.addPropertyChangeListener(l);
    }
    public void removePropertyChangeListener(PropertyChangeListener l) {
        changes.removePropertyChangeListener(l);
    }
    public String getText() {
        return text;
    }
    public void setText(String newText) {
        String oldText = text;
        text = newText;
        changes.firePropertyChange("text", oldText, newText);
    }
}
```

Exemplu PropertyChange (2)

Crearea unui listener

```
public class MyBeanListener implements PropertyChangeListener {  
  
    public void propertyChange(PropertyChangeEvent e) {  
        // reactioneaza la schimbarea textului  
    }  
  
    public MyBeanListener(MyBean bean) {  
        bean.addPropertyChangeListener(this);  
    }  
}
```

Proprietăți constrânse

O proprietate este constransă atunci când schimbarea ei poate fi **refuzată** de către un obiect (chiar de către componenta însăși) care are **drept de veto** asupra acelei proprietăți.

Informarea ascultătorilor cu drept de veto asupra schimbării unei proprietăți se face **înainte** ca schimbarea să aibă efectiv loc, prin intermediul unui eveniment de tip `PropertyChangeEvent`.

Exemplu VetoableChange (1)

```
public class MyBean {
    private String text;
    private VetoableChangeSupport vetos=new VetoableChangeSupport(this);

    public void addVetoableChangeListener(VetoableChangeListener l) {
        vetos.addVetoableChangeListener(l);
    }
    public void removeVetoableChangeListener(VetoableChangeListener l) {
        vetos.removeVetoableChangeListener(l);
    }

    public void setText(String newText) throws PropertyVetoException {
        String oldText = text;
        vetos.fireVetoableChange("text", oldText, newText);
        text = newText;
        changes.firePropertyChange("text", oldText, newText);
    }
}
```

Exemplu VetoableChange (2)

Crearea unui listener

```
public class MyBeanListener implements VetoableChangeListener {

    public void vetoableChange(PropertyChangeEvent e)
        throws PropertyVetoException {
        // drept de veto asupra schimbarii textului
        String prop = e.getPropertyName();
        if (prop.equals("text")) {
            String text = (String)e.getNewValue();
            if (text.length() == 0 ) {
                throw new PropertyVetoException("Textul este obligatoriu!", e);
            }
        }
    }

    public MyBeanListener(MyBean bean) {
        bean.addPropertyChangeListener(this);
    }
}
```

Proprietăți indexate

O proprietate indexată este o proprietate al cărei domeniu de valori este reprezentat de o **mulțime indexată** (tablou).

```
private String items[] = {"bean", "jsp", "servlet"};
public String[] getItems() {
    return items;
}
public void setItems(String items[]) {
    this.items = items;
}
public String getItems(int index) {
    return items[index];
}
public void setItems(String items[], int index) {
    this.items[index] = items[index];
}
```

Introspecția

Intrsopectia este procesul prin care sunt determinate proprietatile, metodele si evenimentele unei componente Bean. Aceasta se face fie prin intermediul clasei **Introspector**, fie direct prin mecanismul de reflectare oferit de Java (Reflection API).

Implicit, clasa `Introspector` foloseste ea insasi mecanismul de reflectare pentru inspectarea componentei, insa acest comportament poate fi schimbat prin construirea unei clase de tip **BeanInfo**.

Crearea unui BeanInfo

Scopul unei clase **BeanInfo** este de a furniza introspectorului suficiente informatii pentru descrierea si personalizarea componentei.

- Expunerea doar a unui subset specificat din proprietatile componentei.
- Specificarea unui nume mai descriptiv pentru proprietati, altul decat cel implicit.
- Despartirea proprietatilor in: normal si expert.
- Specificarea unei clase dedicate pentru personalizarea componentei (customizer)
- Asocierea unei imagini componentei, etc.

Crearea unui PropertyEditor

Scopul unei clase **PropertyEditor** este de a oferi mecanismul necesare pentru editarea proprietăților unui bean în cadrul unui container.

```
public class MyBeanTextEditor
    extends java.beans.PropertyEditorSupport {

    public String getAsText() {
        //Valorea initiala
        return "Hello";
    }
    public void setAsText(String text) throws IllegalArgumentException {
        if (text.length() > 10)
            throw new IllegalArgumentException("Text prea lung!");
        super.setAsText(text);
    }
}
```

Crearea unui Customizer

Un **Customizer** oferă o interfață completă pentru editarea proprietăților unui bean. Aceasta trebuie să fie extinsă din clasa `java.awt.Component` pentru a putea fi instanțiată de container în cadrul unei ferestre specifice.

Asocierea între o componentă bean și o clasă de tip `Customizer` se face în metoda `getBeanDescriptor` a clasei `BeanInfo`

Considerații generale

- Definesc un model pentru crearea de componente
- Pot fi grupate în librării (.jar)
- Pot fi folosite în diferite contexte:
 - **GUI** - de exemplu bean-uri Swing sau AWT
 - **Model** - reprezentarea datelor
- Sunt manevrate de un container, prin intermediul unui introspector
- Trebuie să fie agnostice de modul în care vor fi folosite