Java Beans

Noël De Palma
noel.depalma@inrialpes.fr
Professeur Université de Grenoble
Remerciement X. BLANC

Plan

Objectifs : pourquoi les JavaBeans ?

Utiliser les JavaBeans

Evénements

Propriétés

Persistance

Introspection

Manipuler et déployer les JavaBeans

Objectifs

- Définir un modèle de composant logiciel pour Java
  - Blocs de construction pour composer des applications
  - Des tierces parties peuvent créer des composants Java qui seront assemblés ensemble pour créer des applications pour les usagers finaux
- Architecture plate de composants
- Simplicité

Définition

Un Java Bean est un composant logiciel réutilisable qui peut être manipulé visuellement dans un outil d'assemblage (éditeur graphique, BeanBox...).

- Exemples d'outils d'assemblage
  - Editeur de page Web
  - Editeur visuel d'applications
  - Constructeur de GUI
  - Editeur d'applications serveurs

Caractéristiques distinctives des JavaBean

- Support pour l'*introspection* : un outil d'assemblage puisse analyser comment le bean travaille

- Support pour la "*customization*" : l'utilisateur peut configurer l'apparence et le comportement du bean

- Support pour les *événements* : mécanisme simple de communication qui peut être utilisée pour connecter les beans entre eux.

- Support pour les *propriétés* à des fins de customization et de programmation.

- Support pour la persistence : un bean peut être configuré par un éditeur d'applications et son état configuré peut être sauvé et récupéré plus tard

Principales caractéristiques des JavaBean

- Ensemble de propriétés exposées
  - attributs nommés (variables d'instance)

- Ensemble de méthodes que les autres composants sont autorisés à invoquer
  - par défaut, toutes les méthodes publiques

- Ensemble d'événements déclenchés
  - une façon d'informer les autres composants que quelque chose d'intéressant est survenu

JavaBeans en tant que lien à un état distant (remote state)

- Accés local
  - le modèle d'exécution de base est qu'un bean s'exécute dans le même espace d'adresse que son contenant
- Mécanismes principaux d'accès au réseau
  - Java RMI
  - Java IDL: modèle d'objet distribué de CORBA (OMG)
  - Au travers du WEB via un serveur d'application

Java Beans

- Construction d'un Beans
  - Modèle Beans
  - Règles syntaxiques
  - Package
- Déploiement d'un Beans
  - Utilisation d'outils graphiques
- Exécution d'un Beans
  - programme classique

Construction d'un Beans

Modèle Beans et Règles de syntaxes

Modèle Beans

- Un Bean est une classe
- Cette classe doit être conforme au modèle Beans
- Ce modèle permet aux outils de découvrir les Beans
- Un Bean encapsule sa propre description
  =>Propriété générale des COMPOSANTS

Modèle Beans

- Un Bean est une classe Java
- Peut implanter l'interface java.io.Serializable
- Fournir un constructeur public sans argument
- Possède des propriétés
- Peut envoyer des événements

Méthodes du bean

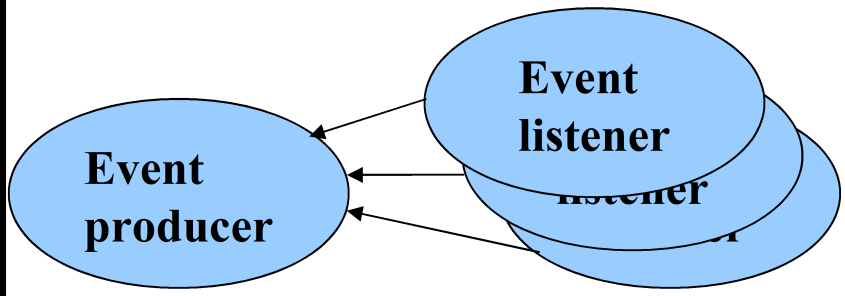
- Pour qu'un bean puisse offrir ces services, il faut que les méthodes soient public.

  public type nom_methode(parametre) {…}

Evenements

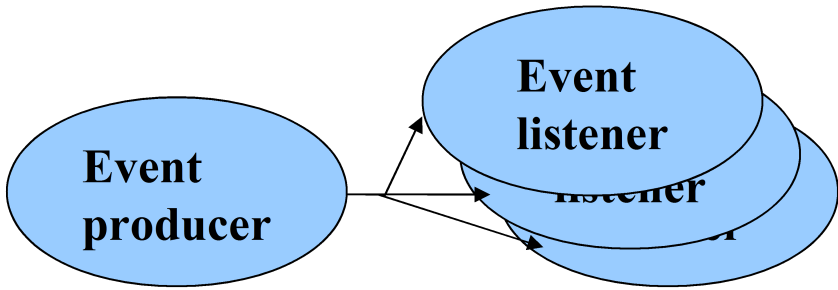
- Suit le modèle événementiel de Java
- Mécanisme de communication de type publish/subscribe
  - Event Producer (producteur d'événement)
  - Event Listener (écouteur d'événement)
  - L'émission d'un événement est appelé une notification
  - Communication annonyme 1=>N

Subscribe Phase



addXXXListener

Publish Phase



Callback

Evenements

- Intérêt du publish/subscribe
  - Désignation anonyme
  - => meilleurs reutilisabilité
- Attention !!!
  - Un listener peut recevoir des notifications d'événement en parallèle.
  - Les méthodes de notification devraient être synchronized

Evenements

- **Événement**
  - Convention de nom : **XXXEvent (ex mouseEvent)**
  - Sous classe de **java.util.EventObject**
  - Un événement comporte toujours la référence de l'objet producteur
- **Listeners**
  - Notifier un événement => appel d'un callBack pour chaque écouteurs
  - Définie dans une interface **XXXListener** (Ex : MouseListener)
    - Cette interface doit étendre **EventListener**
  - Un listener pour l'événement **XXX** doit implanter **XXXListener**
    - **Ex : MouseListener**
  - Un listener (L) doit s'enregistrer auprès du producteur d'événement (P)
    - **P.addXXXListener(L) // ex P.addMouseListener**
  - En pratique utilisation de classes abstraites ou de classe anonyme comme base pour les écouteurs

Evenements

- Producteur source d'un événement XXX
  - Implante 2 méthodes :

```
public void addXXXListener(XXXListener obj)
public void removeXXXListener(XXXListener obj)
```

- Notification d'un événement
  - Parcourir la liste des auditeurs et appeler la méthode de callback
- Cas particulier des propriétés liées
  - PropertyChangeSupport (voir plus loin)

Sémantique de la livraison des événements

- **Livraison synchrone**
  - **La livraison des événements est synchrone par rapport à la source des événements.**

- **Exceptions**
- **les méthodes des listeners peuvent déclencher des exceptions déclarées**
- **les sources d'événements doivent être conscientes que les listeners peuvent déclenchés "par inadvertance" des exceptions non déclarées.**

  - **si un des listeners de l'exception déclenche une exception :**
    - **c'est une décision de l'implémentation au niveau de la source des événements de décider si elle poursuit ou non la livraison de l'événement aux autres listeners**

Propriétés du Bean

- Un Beans possède des propriétés
- Les propriétés d'un Beans sont les aspects du Beans qui pourront être modifier lors du déploiement du Bean
- Ex: couleurDeFond, typeDeFont ...

Property Editors / Customizers

- Les Beans sont manipulés par un outil graphique. Cet outil va permettre de saisir les valeurs de bases des propriétés.
- Les Property Editors permettent de paramètrer la façon de saisir les informations dans l'outil graphique.
- Les Customizers sont des composants graphiques qui vont être intégrés dans l'outil pour la saisie des propriétés.

Propriétés Simples

- L'accès aux propriétés se fait par des méthodes set / get.
- Example :

```
private int toto; // La propriete
public int getToto() { return toto;} //La syntaxe doit être respectée
public void setToto(int value) { toto=value;} //idem
```

- Ces règles syntaxiques permettent aux outils de découvrir les méthodes.

Exemple de beans—propriétés simples

```
public class BeanCompteur implements java.io.Serializable{
    private int accu=0; /** Holds value of property accu. */
    private int pas = 1; /** Holds value of property pas. */

    public BeanCompteur() {}

    public int getAccu() { return this.accu;}
    public void setAccu(int val) {this.accu = val;}
    public int getPas() {return this.pas;}

    public void incremente() {setAccu(accu+pas);}
}
```

Propriétés Indexées

- Pour spécifier un indice qui identifie quelle valeur manipuler
- Les indices de propriétés doivent être de type `int` (Java).
  - Cette contrainte sera relaxée dans le futur pour permettre d'autres types d'indice.
- Les Getter/Setter indexées

```
public void setXXX(int index, type value);
public void setXXX(type values[]);
public type getXXX(int index);
public type[] getXXX();
```

- Peut déclencher une exception

`java.lang.ArrayIndexOutOfBoundsException`

Propriétés Liées

- **Les propriétés peuvent être liées: un changement d'une propriété d'un Bean entraîne un changement sur un autre Bean.**
- Le lien entre deux propriétés est unidirectionnel. Il y a donc la propriété dépendante et la propriété lié.
- En fait, il y a la propriété lié et le bean qui contient la propriété dépendante.
- L 'utilisation d'une propriété liée consiste à notifier divers événements aux beans intéressés.
- Utilisation de *propertyChangeSupport* qui simplifie la gestion des listeners

Propriétés liées

- S'il existe un service de notification des changement d'une propriété ➔ bound

- *PropertyChangeListener* event listener
  - pour informer des mises-à-jour des propriétés liées simples.

```
public void addPropertyChangeListener
        (PropertyChangeListener x);
public void removePropertyChangeListener
        (PropertyChangeListener x);
```

  - invocation de la méthode suivante pour chacun des

```
aListener.propertyChange(PropertyChangeEvent
  evt)
```

Propriétés Liées

- Le Bean contenant la propriété liée

```
import java.beans.*;
...
private PropertyChangeSupport changes = new PropertyChangeSupport(this);

public void addPropertyChangeListener( PropertyChangeListener l) {
      changes.addPropertyChangeListener(l);
}

public void removePropertyChangeListener(PropertyChangeListener l) {
     changes.removePropertyChangeListener(l);
}

public void setLabel(String newLabel) {
   String oldLabel = label;
   label = newLabel;
   ...
   changes.firePropertyChange("label", oldLabel, newLabel);
}
...
```

Propriétés Liées

- L' écouteur de Bean

```
public class MyClass implements java.beans.PropertyChangeListener, java.io.Serializable { …

    public void propertyChange(PropertyChangeEvent evt) {…}

      ...
    }
```

- L'écouteur de bean doit être enregistré auprés du producteur

```
MyClass m = new Myclass();
Prod.addPropertyChangeListener(m);
```

- Possibilité d'utiliser des classes anonymes

Exemple propriétés liées (bean compteur suite)

```
public class BeanCompteur implements java.io.Serializable{
 private PropertyChangeSupport chg=new PropertyChangeSupport(this);
 private int accu=0;
 private int pas = 1;
 public BeanCompteur() {}
 public void addPropertyChangeListener(PropertyChangeListener l{
     chg.addPropertyChangeListener(l);}
 public void removePropertyChangeListener(PropertyChangeListener l){
        chg.removePropertyChangeListener(l);}
  public int getAccu() { return this.accu;}
  public void setAccu(int val) {
        this.accu = val;
       chg.firePropertyChange ("accu", new Integer (oldAccu), new Integer
(accu));}
   ...
}
```

Propriétés liées

Example pour une Applet écrite à la main avec un classe anonyme

```
package packDes1;
import java.beans.*;
public class AppletACCU extends java.applet.Applet {
private packDes1.BeanCompteur cpt;
private java.awt.Label valAccu;

public void init() {
  cpt = new packDes1.BeanCompteur();
  valAccu = new java.awt.Label();
  cpt.addPropertyChangeListener(new PropertyChangeListener(){
     public void propertyChange(
             java.beans.PropertyChangeEvent evt) {
      valAccu.setText(""+cpt.getAccu());
    }
  });
...
```

Propriétés Contraintes

- Une propriété de Beans peut être contrainte
  - Consensus des Listerners
- Les propriétés sont dites contraintes lorsque d'autres beans peuvent
  - souhaiter valider le changement
  - rejeter un changement s'il est inapproprié
- Les contraintes sont un cas particulier des propriétés liées, un bean peu émettre un veto sur le changement d'une propriété contrainte.
- Les mécanismes de propriété contrainte sont similaires aux propriétés liées avec une possibilité de véto

Propriétés contraintes

- Si le récepteur de l'événement ne souhaite pas que la modification soit faite

  → il déclenche une exception `PropertyVetoException`

- la responsabilité de la source de
  - capturer cette exception
  - réétablir l'ancienne valeur
  - rapporte le retour l'ancienne valeur via un nouvel événement `VetoableChangeListener.vetoableChange`.

- La source devrait déclencher ce type d'événement **avant** de mettre à jour son état.

Propriétés contraintes

- Il existe une classe utilitaire `VetoableChangeSupport`
  - pour gérer la liste des `VetoableChangeListener`
  - pour déclencher les événements `VetoableChange`
  - pour capturer les exceptions `PropertyVetoException` et émettre les événements de réversion nécessaires.

Ecouter les propriétés à la fois liées et contraintes

Si un bean supporte une propriété qui est à la fois liée et contrainte

alors

elle devrait déclencher

- un événement `VetoableChangeListener.vetoableChange` avant la mise-à-jour de la propriété
- un événement `PropertyChangeListener.propertyChange` après la mise-à-jour de la propriété.

Propriétés Contraintes

- Le Beans qui contient la propriété contrainte

```
import java.beans.*;
private VetoableChangeSupport vetos = new VetoableChangeSupport(this);

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

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

public void setPriceInCents(int newPriceInCents) throws PropertyVetoException {
  int oldPriceInCents = ourPriceInCents;
  vetos.fireVetoableChange("priceInCents",new Integer(oldPriceInCents),
            new Integer(newPriceInCents));
  ourPriceInCents = newPriceInCents;
  changes.firePropertyChange("priceInCents",new Integer(oldPriceInCents),
  new Integer(newPriceInCents));
}
```

Propriétés Contraintes

- Beans émettant le veto

```
public class MyClass implements VetoableChangeListener  {
  ….
  void vetoableChange(PropertyChangeEvent evt) throws PropertyVetoException {
        ….
  }
}
```

Résumé : Les classes de support en Java

- class java.beans.**PropertyChangeEvent**

- interface **java.beans.PropertyChangeListener**
- class **java.beans.PropertyChangeSupport**
  - Aucun événement n'est déclenché si les anciennes valeurs sont égales et non-nulles.

- class **java.beans.PropertyVetoException**
- interface **java.beans.VetoableChangeListener**
- class **java.beans.VetoableChangeSupport**

Persistance

- Les beans sont des objets persistant
  - Objet dont on peut sauver/restaurer l'état sous la forme d'octets
  - L'état peut être durable (Beans dans un fichier sauvé sur disque)
  - L'état peut être transmis sur le réseau (Bean dans une applet)
- La persistance des Bean est basée sur la sérialization JAVA

Serialization

- Objets serializable
  - Type de base
  - Objets qui implémente serializable (interface vide) et dont les attributs sont serializables. Si un attributs n'est pas serializable, il doit être transient.
  - Tous les objets ne sont pas serializable (file, thread, socket ...)
  - Peuvent être transmis lors d'entrées/sorties (disque, réseau …)
  - Méthodes readObject/writeObject par défaut
- Les propriétés d'un bean doivent donc être des types de bases ou serializable si le bean est serializable

Flux d'entrée/sortie

- Deux (principales) classes abstraites
  - OutputStream : flux d'octets en sortie
  - InputStream : flux d'octets en entrée
- Plusieurs sous classes en fonction de la destination/source des octets (fichier …)
  - Ex : FileOutputStream/FileInputStream …
- Plusieurs classes pour le format des données sur le flux d'entrée/sortie
  - DataOutputStream/DataInputStream (types primitifs)
  - ObjectOutputStream/ObjectInputStream (objets serializable)

Exemple serialization et E/S

```
// Exemple Bean voiture
public class voiture implements java.io.Serializable{
   private String type;
   private int imm;

   public voiture() {};

   public String getType(){ return type;}
   public void setType(String t){ type=t;}

   public int getImm(){ return imm;}
   public void setImm(int ima){ imm=ima;}
}
```

Exemple serialization et E/S

```
// exemple programme de serialization de bean
Public class save {
  Public static void main (String args[]){
     Voiture v = new Voiture();
     v.setImm(123454538);
     v.setType("Peugeot");
     Try {
       FileOutputStream f=new FileOutputStream("voit1.bin");
       ObjectOutputStream s=new ObjectOutputStream(f);
       s.write(v);
       s.close();
     } catch (Exception e){e.printStackTrace();}
  }
}
```

Exemple serialization et E/S

```
// exemple programme de deserialization de bean
Public class load {
  Public static void main (String args[]){
     Try {
       FileInputStream f=new FileInputStream("voit1.bin");
       ObjectInputStream s=new ObjectInputStream(f);
       Voiture v = s.read();
       s.close();
       System.out.println("Bean deserializé : "+v.getImm()+" "+v.getType());
     } catch (Exception e){e.printStackTrace();}
  }
}
```

Introspection

Comment connaître un Java Bean

Principe

- Un bean est un composant qui va être manipulé par des outils graphiques.
- Le bean doit donc permettre à ces outils de savoir comment le bean fonctionne.
- L'introspection permet de connaître les propriétés, les méthodes et les événements que manipule le bean.

Comment fonctionne l'introspection?

- Par défaut
  - Un mécanisme réflexif de bas-niveau
  - Utilisation de *design patterns* simples pour déduire à partir des méthodes quels sont les propriétés, les événements et les méthodes publiques qui sont disponibles.
    - *design patterns* = noms et signatures stéréotypés pour un ensemble de méthodes et/ou d'interfaces
- L'interface BeanInfo
  - pour exercer un contrôle complet et précis sur les propriétés, les événements et les méthodes qui sont exposées.

Le package reflect

- Java propose une API pur découvrir une classe lors de l'exécution.
- Ce qui permet d'exécuter dynamiquement une méthode sur une classe.
- Le package reflect peut être utilisé pour n 'importe quelle classe Java, donc aussi pour les bean.

La classe Introspector

- Le package java.beans propose une autre manière de découvrir une classe
- Cette manière se fait à travers la classe Introspector.
- L'introspection de Java Bean utilise la réflexion mais aussi les règles syntaxiques de Java Beans
- L'introspection permet aussi d'utiliser la classe BeanInfo pour décrire un Java Bean

Analyse d'un bean

- La classe java.beans.Introspector
  - traverse la hiérarchie des classes / superclasses d'une classe donnée.
  - A chaque niveau
    - Vérifie s'il existe une spécification explicite (BeanInfo)
    - Si non, effectue une analyse implicite fondée sur les design patterns (getXXX, setXXX …)

Si

- une classe fournit un BeanInfo explicite sur elle-même

alors

- l'information fournie par le BeanInfo est ajoutée à l'information obtenue par l'analyse des sous-classes

- cette information explicite est considérée comme finale pour la classe courante et ses superclasses,

- l'introspection s'arrête et ne va pas plus loin dans la hiérarchie des superclasses

Obtenir la référence vers un BeanInfo

- Si aucun BeanInfo n'a été proposé par le constructeur de bean, il en sera généré un automatiquement.

```
MonBean mb = new MonBean();
BeanInfo bi = Introspector.getBeanInfo(mb.getClass());
```

La classe BeanInfo

- L'interface BeanInfo
  - les méthodes permettent de connaître
    - événements
    - propriétés
    - méthodes
    - information globale
- Un objet de type BeanInfo est retourné lors de l'appel à l'introspector.
- Si, le bean possède un BeanInfo qui lui est propre, cette objet sera de ce type.

Des méthodes de la classe BeanInfo

- public MethodDescriptor[] getMethodDescriptors()
- public PropertyDescriptor[] getPropertyDescriptors()
- public EventSetDescriptor[] getEventSetDescriptors()
- public BeanDescriptor getBeanDescriptor()

Des méthodes de PropertyDescriptor

- getPropertyType()
- isConstrained()
- GetName());
- getReadMethod());
- getWriteMethod())

Création d'un BeanInfo

- Le nom de la classe BeanInfo doit être le nom du bean précédé de BeanInfo
  Ex: MonBeanBeanInfo
- Cette classe doit hériter de SimpleBeanInfo
- Re-écrire les méthodes que l'on veut modifier.

Exemple : getPropertyDescriptors()

```
PropertyDescriptor[] proDesc = bi.getPropertyDescriptors();
  for (int i=0; i<propDesc.length; i++) {
    System.out.println("Nom propriete :"+  propDesc[i].getName());
    System.out.println("Type propriete :"+ propDesc[i].getPropertyType());
    System.out.println(" Getter propriete :"+ propDesc[i].getReadMethod());
    System.out.println(" Setter propriete :"+ propDesc[i].getWriteMethod());
}
```

Exemple : getMethodDescriptors()

```
MethodDescriptor[] methodDescriptor;
unMethodDescriptor = bi.getMethodDescriptors();
for (int i=0; i < unMethodDescriptor.length; i++) {
  System.out.println(" Methode :
    "+unMethodDescriptor[i].getName());
}
```

Exemple : getEventSetDescriptors()

```
EventSetDescriptor[] unEventSetDesc = bi.getEventSetDescriptors();
for (int i = 0; i < unEventSetDesc.length; i++) {
  System.out.println(" Nom evt :" + unEventSetDesc[i].getName());
  System.out.println(" Methode add evt     : " +
        unEventSetDesc[i].getAddListenerMethod());
  System.out.println(" Methode remove evt  : " +
      unEventSetDesc[i].getRemoveListenerMethod());
  unMethodDesc = unEventSetDesc[i].getListenerMethodDescriptors();
  for (int j = 0; j < unMethodDesc.length; j++) {
      System.out.println(" Event Type: " + unMethodDesc[j].getName());
  }
}
```

Déploiement de Beans

Utilisation de la Bean Box

Principe

- Nous avons vu qu'un bean était constitué de plusieurs classe Java.
- Cependant, ces classes forment un tout : le bean.
- Voilà pourquoi il faut archiver toutes les classes dans un seul fichier. Un fichier .jar

Le manifest du fichier Jar

- Dans tout archive .jar, il y a un fichier manifest.mnf qui décrit le contenu de l'archive.
- Pour les Java Beans le manifest doit contenir:

  Name: toto.class

  Java-Bean: True

Le fichier Jar

- Pour archiver des classes dans un fichier Jar:

  jar cfm fichier.jar fichier.mnf liste_fichier

  ex:jar cfm toto.jar toto.mnf *.class

Conclusion JavaBeans

Modèle de composants

- intégration au niveau source (Java)
- modèle simple et commode pour les composants individuels
- modèle bien adapté aux composants graphiques
- outils limités pour la description de l'architecture
  - pas de visualisation globale de l'architecture

Environnement de développement (BeanBox)

- outils d'aide à la construction
  - édition simple des propriétés et des méthodes
  - plus dificile pour les liens entre Beans
- non prévu pour la répartition

Conclusion Java Bean

- Interface
  - Méthodes, propriétés, source d'événement, listeners (puits d'événement)
- Implementation
  - Une classe
- Assemblage
  - Événements et listeners.
  - Bean box. Graphique. Aide dynamique à la conception.
- Cycle de vie
  - jar pour le déploiement
- Infrastructure
  - Une JVM!

Référence

[Sun, 1997] Sun Microystem, JavaBeans, Version 1.01, Graham Hamilton (Editor), July 24, 1997, http://java.sun.com/beans/beans.101.pdf

http://wiki.netbeans.org/NetBeansJavaBeansTutorial