

Guide du Développeur JavaBean

Ce document présente de manière synthétique les exigences et bonnes pratiques du cadre de développement des JavaBeans. Il est fourni tel quel comme aide pour l'élaboration de vos Beans tout au long des TDs consacrés aux composants.

LA CLASSE SOURCE (LE BEAN)

Rôle

Il s'agit de la classe de base du composant (unique, bien qu'elle puisse elle-même utiliser [via la composition] d'autres classes). Autant que possible on la dérivera d'une classe générale, particulièrement pour les composants graphiques (JPanel fournit une bonne base). Elle fournit tous les services qui seront exposés par le composant.

Contraintes de base

Afin d'être reconnue comme un bean par les divers environnements, la classe doit respecter quelques règles :

- Etre publique
- Fournir un constructeur public sans paramètre (bien que d'autres constructeurs puissent être fournis, les « bonnes pratiques » recommandent de n'en rien faire).
- Implémenter directement ou indirectement (via l'héritage) l'interface **Serializable**. A noter que cette interface ne définit aucune méthode (il s'agit d'une interface dite de « tagging », c'est-à-dire dont le rôle est uniquement d'indiquer que les objets de la classe peuvent être sauvegardés).

On demande également que ces méthodes soient « Thread-safe », c'est-à-dire qu'elles fonctionnent correctement dans un environnement où coexistent plusieurs threads. On utilisera (si nécessaire) à cette fin le mot clé **synchronized** (placé devant une méthode ou un bloc) qui permet de garantir que la fonction ou le bloc concerné ne sera jamais accédé par plus d'un processus en même temps. Il va de soi que cette sécurité à un coût, celui du temps perdu par les processus attendant que la ressource se libère.

Propriétés

Les propriétés sont les attributs qui seront exposés par le composant. Il s'agit en fait non d'attributs publics (ce qui serait contraire au principe de l'encapsulation) mais bien de couples de fonctions (getter/setter) publics en rapport avec un ou plusieurs attributs privés. Ces méthodes donneront une sorte d'illusion d'accès public à un attribut privé tout en conservant les possibilités de vérification à l'entrée ou de formatage à la sortie.

On distingue les propriétés *simples*, qui accèdent à ou modifient un élément unique des propriétés *indexées*, qui concernent des éléments indexés (tableaux, vecteurs...).

L'exposition des propriétés se fait par le mécanisme **d'introspection** : l'environnement dans lequel le Bean est chargé va examiner les noms de méthode du Bean pour déterminer

implicitement les propriétés disponibles. Afin que ce mécanisme puisse fonctionner correctement, il est nécessaire d'utiliser certains patterns de noms pour les propriétés.

1 Propriétés simples

Si la propriété se nomme `prop` et est de type `Type`, les signatures de fonction devront être de la forme suivante :

```
Type getProp(); //accesseur
void setProp(Type value); //modificateur
```

Exception : pour des propriétés de type `boolean`, on utilisera pour l'accesseur la forme :

```
boolean isProp() //accesseur pour un boolean
```

2 Propriétés indexées (tableaux)

Si la propriété se nomme `prop` et est de type `Type`, on trouvera quatre fonctions : deux pour l'accès à une variable, et deux pour l'accès à l'ensemble des variables. Les signatures de ces fonctions devront être de la forme suivante :

```
Type getProp(int index) // accesseur unique
void setProp(int index, Type1 value) // modificateur unique
Type[] getProp() // accesseur pour l'ensemble
void setProp(Type[] values) // modificateur pour tout
```

La remarque pour les `boolean` est également de mise ici.

Propriétés liées et propriétés à veto

1 Propriété liée (« bounded property »)

Propriété pour laquelle tout changement de valeur entraîne automatiquement la génération et l'envoi des deux valeurs (l'ancienne et la nouvelle) à tous les écouteurs inscrits. Ceux-ci peuvent alors y réagir immédiatement de manière adéquate.

2 Propriété à veto (« constrained property »)

Suit le même principe, mais en poussant les choses plus loin : lorsqu'un changement de valeur doit intervenir, tout écouteur inscrit peut y mettre son veto avant qu'il ait effectivement lieu. Ceci permet de vérifier le respect de certaines conditions externes à la classe avant de permettre le changement.

Si le mécanisme mis en œuvre dans les deux cas est très proche de celui de la gestion d'événements, il est grandement simplifié par l'utilisation de deux classes d'aide de l'API `JavaBeans`, à savoir `PropertyChangeSupport` (pour les propriétés liées) et `VetoableChangeSupport` (pour les propriétés à veto). Ces deux classes fournissent, outre un constructeur, les méthodes permettant d'ajouter ou de retirer un écouteur, ainsi qu'une méthode envoyant le nom de la propriété, son ancienne valeur et la nouvelle (proposition, dans le cas d'une propriété à veto) à tous les écouteurs inscrits.

```
void addPropertyChangeListener(PropertyChangeListener p)
void removePropertyChangeListener(PropertyChangeListener p)
void firePropertyChange(String n, Object oldValue, Object newValue)
```

(remplacer `Property` par `Vetoable` pour `VetoableChangeSupport`)

Les écouteurs devront eux simplement implémenter soit `PropertyChangeListener`, soit `VetoableChangeListener`. Un écouteur d'une propriété à veto signifiera son refus en levant une `PropertyVetoException`.

L'objet source doit bien évidemment intégrer correctement la présence d'écouteurs pouvant éventuellement poser leur veto à une modification de propriété. A cette fin, on respectera la démarche suivante : lorsqu'un changement de valeur est proposé, cette proposition est envoyée aux écouteurs à veto. Si aucune exception ne se produit (c'est-à-dire qu'aucun écouteur ne pose son veto à la proposition), la valeur de la propriété est alors (et alors seulement) effectivement modifiée. Enfin, une notification de cette modification est envoyée aux écouteurs sans veto.

Gestion d'événements

Pour qu'un Bean (ou de manière générale une classe) puisse jouer correctement son rôle de source d'événements, il lui faut d'une part gérer une liste d'écouteurs inscrits, et d'autre part créer et envoyer un événement à tous les inscrits quand certaines conditions sont remplies.

La gestion de la liste se fait typiquement via la définition d'un couple de fonctions du type

```
void addEventListener(EventTypeListener e)
void removeEventListener(EventTypeListener e)
```

(en remplaçant EventType par le nom correct de l'événement) :

En interne, on utilisera en général une `Collection` pour implémenter la liste stockant les références des différents écouteurs. `Vector`, bien que plus lent que `ArrayList`, est à préférer dans ce cas car toutes ses méthodes sensibles sont **synchronized** et donc « Thread-safe ».

L'envoi d'un événement consiste simplement en la création d'un objet de la classe d'événements ad hoc et de son envoi à tous les écouteurs inscrits via un parcours du vecteur. On appellera donc pour chaque inscrit la méthode déclarée dans l'interface de l'écouteur, prenant comme paramètre l'événement créé.

LES CLASSES ECOUTEURS (LISTENERS)

Le but est évidemment que les différents écouteurs puissent être des classes pratiquement quelconques. On représentera donc le contrat d'un écouteur sous forme d'une interface dont le nom sera par convention « `EventListener` ». Cette interface devra déclarer la ou les méthodes par lesquelles une source indiquera à un écouteur qu'un événement s'est produit. Ces méthodes prendront évidemment comme paramètre au minimum un objet du type d'événement concerné.

L'interface devra hériter de l'interface `java.util.EventListener` (interface dite de « tagging »), ne déclarant aucune méthode, mais permettant aux IDE de reconnaître l'interface comme un écouteur).

Exemple : `ActionListener` déclare la méthode suivante :

```
void actionPerformed(ActionEvent e)
```

Toutes les classes d'écouteur implémenteront l'interface, et définiront l'ensemble des méthodes concernées.

LES CLASSES ÉVÉNEMENTS (EVENTS)

Pour qu'un objet soit reconnu comme un événement, il doit hériter de la classe `java.util.EventObject`. Le constructeur de `EventObject` prend en paramètre l'objet source lui-même (accessible via la méthode `getSource()`). La classe pourra déclarer sous forme d'attributs et de méthodes toute information complémentaire que l'objet source pourrait souhaiter communiquer à ses écouteurs. Un objet événement est créé par l'objet source et passé comme paramètre de la méthode d'écoute à tous ses écouteurs.

LA CLASSE D'INFORMATION (BEANINFO)

La classe d'information sert à compléter ou corriger l'information que les divers environnements de développement obtiennent du bean par introspection.

Informations supplémentaires

Pour ce qui est des informations supplémentaires, il s'agit essentiellement :

- D'un set d'icônes permettant de représenter le bean dans une palette.
- De la possibilité de définir un événement et une propriété « par défaut », à savoir ceux qui seront logiquement les plus utilisés par un humain. Les divers environnements auront alors la possibilité de mettre cet événement et/ou cette propriété « en avant » (en la proposant par défaut via l' « auto-complete », par exemple).

Corrections

Les corrections viennent en fait essentiellement de la possibilité de ne pas publier l'ensemble des propriétés, événements ou méthodes du composant. L'objectif est ici de ne pas noyer l'utilisateur. Un simple exemple permet de se rendre compte de l'utilité d'une telle fonctionnalité : un composant héritant de `JPanel` hérite donc indirectement de `JComponent`, `Container`, `Component` et `Object`, et ainsi de 12 événements, une quarantaine de propriétés et plus de 150 méthodes ! Il va de soi que seul un sous-ensemble réduit d'entre elles intéresse l'utilisateur du composant.

La classe d'information présente donc trois méthodes permettant d'obtenir respectivement les événements, méthodes et propriétés publiées du composant. Si l'une de ces méthodes renvoie la valeur `null`, c'est l'introspection qui entre en jeu, et l'ensemble concerné est alors intégralement publié.

Pratiquement, une classe d'information se présente sous forme d'une classe portant le même nom que le bean avec le suffixe « `BeanInfo` ». Un composant « `Comp` » aura donc une classe d'information nommée « `CompBeanInfo` ». Cette classe doit impérativement implémenter l'interface `java.beans.BeanInfo` [API] (et donc toutes les méthodes qui y sont définies).

Afin de simplifier le travail du créateur de composant, une classe d'aide nommée `java.beans.SimpleBeanInfo` [API] a été créée. Elle implémente l'interface de la manière la plus simple qui soit : en renvoyant la valeur `null` à toutes ses méthodes. Le concepteur d'une classe d'information peut dès lors la faire hériter de `SimpleBeanInfo`, et ne redéfinir que les méthodes où il ne souhaite pas voir le processus d'introspection se mettre en place.

Les principales méthodes de l'interface sont reprises ici :

```
EventSetDescriptor[] getEventSetDescriptors()  
PropertyDescriptor[] getPropertyDescriptors()  
MethodsDescriptor[] getMethodDescriptors()
```

Ces trois méthodes renvoient respectivement les événements, propriétés et méthodes du composant devant être publiées.

```
int getDefaultEventIndex()  
int getDefaultPropertyIndex()
```

Ces deux méthodes renvoient l'index de l'événement et de la propriété par défaut dans leurs tableaux respectifs.

```
BeanDescriptor getBeanDescriptor()
```

Renvoie une information générale sur le bean, constituée du nom de la classe principale du composant, ainsi que de son éventuel « Customizer » (une classe graphique permettant de paramétrer un composant de manière spécifique).

```
Image getIcon(int IconKind)
```

Renvoie une des icônes du composant (en format gif). Quatre icônes différentes devraient être prévues : une 16 * 16 noir et blanc, une 16 * 16 couleur, une 32 * 32 noir et blanc et une 32 * 32 couleur. Le chargement des images à partir des fichiers peut se faire facilement à l'aide de la méthode `loadImage(String)` présente dans la classe `SimpleBeanInfo`.