L'approche Composants Logiciels

Plan

- Pourquoi les Java Beans
- Les concepts de base des beans :
  - propriétés
  - événements
  - action
- Utilisation des beans : adaptateurs et fichiers .jar
- Persistance
- Réflexion et introspection
- Propriétés liées et propriétés contraintes
- Editeurs personnalisés et configurateurs

Introduction

- Bibliographie :
  http://java.sun.com/beans/javadoc/java.beans.SimpleBeanInfo.html
- Idée de base :
  modulariser au maximum un développement pour :
  - réutilisation
  - « localité » (des erreurs notamment)

Modularité : historique

- Premiers pas :
  - structurer le code :
    - notion de fonction/procédure
    - fichiers séparés ⇒ compilation incrémentale et modulaire
    - séparer implantation et interface (fichiers « .c » et « .h »)
    - regrouper des fichiers en forte interaction
      - notion de paquetage (package) : pour la forme source (ADA)
      - notion de bibliothèque (librairy) : pour la forme compilée
  - Structurer les données :
    - structures (C) et enregistrements (Pascal)

L'approche objet

- Premiers langages : Simula, Modula−2
- Buts :
  - structurer le code en fonction des données
  - masquer la représentation des données pour l'évolutivité
  - faciliter la réutilisation au moyen de :
    - l'héritage
    - la généricité
- Principaux succès
  Smalltalk
  C++
  Java

Limites de l'approche objet

- Multiplication des classes rend un développement difficile à comprendre
- Mélange des paquetages et des classes (ADA95, Java) pas toujours très clair
- Classes doivent être conçues les unes en fonction des autres
- Travail de développement = travail d'expert

Les Java Beans

- Définition
  Un Bean = un composant (une classe) réutilisable
- Conséquences :
  - une documentation
  - des propriétés éventuellement modifiables
  - des événements susceptibles d'être générés
  - des actions possibles
- Qu'est−ce−qu'une application :
  ensemble de composants instanciés reliés entre eux
- Qu'est−ce−qu'un lien ?
  Association action d'un composant − événement d'un autre

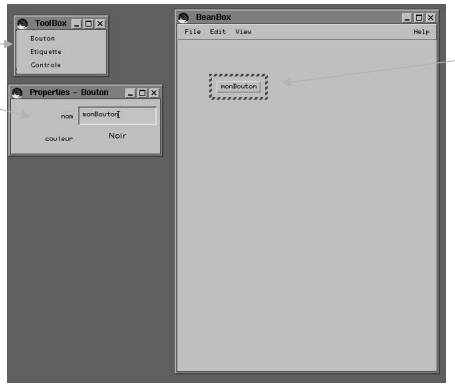
Programmer avec les Beans

- Nécessite un AGL permettant :
  - d'instancier des beans en spécifiant des valeurs pour leurs propriétés
  - d'associer à des événements émis par des beans des actions d'autres beans
  - de sauver les instances et leurs valeurs de propriétés, ainsi que les liens entre elles

  ⇒ exemple (jouet) d'un tel AGL : la *beanbox*
- Conséquences sur les beans :

  doivent respecter certaines contraintes pour que tout AGL désirant les rendre utilisables puisse :
  - connaître et éditer (afficher/modifier) leurs propriétés,
  - connaître les événements qu'ils peuvent émettre
  - connaître les actions qu'ils peuvent effectuer

Beanbox : exemple (1)

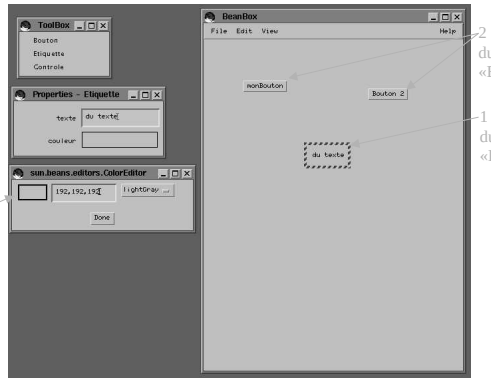


Les beans disponibles

Valeurs des propriétés de l'instance sélectionnée

Une instance du bean « Bouton »

Beanbox : exemple (2)

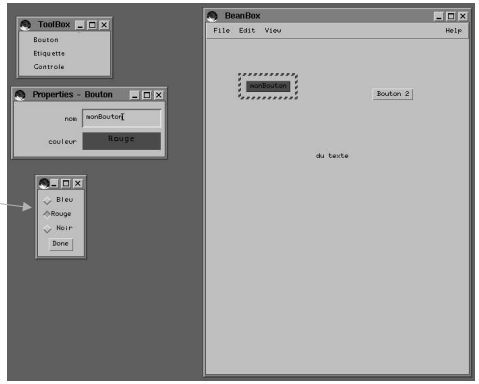


2 instances du bean «Bouton»

1 instance du bean «Etiquette»

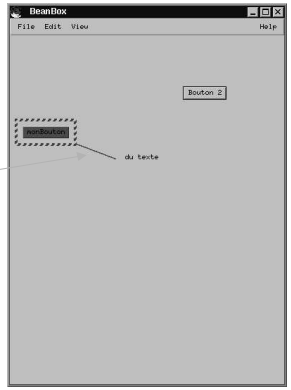
Editeur de propriété par défaut pour les propriétés de type « Color »

Beanbox : exemple (3)



Un éditeur de propriété sur mesure

Beanbox : exemple (4)



Après avoir sélectionné « Mouseclicked » dans la liste des événements générés par le bouton « monBouton », on choisit comme cible l 'instance du bean Etiquette.

Beanbox : exemple (5)

Lorsque l'utilisateur clique sur le bouton « monBouton », c'est la méthode « direKO » de l'étiquette qui sera exécutée.

A partir de maintenant, si l'on clique sur le bouton « monBouton », le texte affiché dans l'étiquette sera « KO ».

Exemple : énoncé

- Les données :

Un composant Etiquette

Propriétés	Actions
couleur de fond	direOK
texte	direKO

Un composant Bouton

Propriétés	Evénement
couleur de fond	ActionEvent
texte	

- Le problème :
  - un bouton bleu « Valider » qui fait afficher « OK »
  - un bouton rouge « Annuler » qui fait afficher « KO »
  - affichages dans une zone de texte à fond jaune

Exemple : solution

- Instanciation des composants
  - une instance IE de Etiquette, couleur fixée à jaune
  - une instance IBV de Bouton dont on fixe la couleur à bleu et le texte à « Valider »
  - une instance IBA de Bouton dont on fixe la couleur à rouge et le texte à « Annuler »
- Les liens
  - association de l'action direOK de IE à l'événement ActionEvent de IBV
  - association de l'action direKO de IE à l'événement ActionEvent de IBR

Propriétés simples d'un Bean

Caractéristiques d'une propriété :
- un nom *nomPropriété*
- un type *typePropriété*
- une éventuelle méthode d'accès en lecture :
  public *typePropriété* getN*omPropriété* ()
- une éventuelle méthode d'accès en écriture :
  public void setN*omPropriété*(*typePropriété* valeur)

- Remarque :
  dans le cas d'une propriété de type booléen, il est possible de remplacer la méthode get*NomPropriété*() par la méthode is*NomPropriété*()

Application à l'exemple (1)

Composant Etiquette

Rappel : le composant Etiquette doit avoir deux propriétés « couleur » et « texte ».

```
import java.awt.*;
public class Etiquette extends Panel {
private Label etiquette;
public Etiquette {etiquette = new Label(); add(etiquette);
  setVisible(true);}
public void setCouleur(Color c) {etiquette.setBackground(c);}
public Color getCouleur() {return etiquette.getBackground();}
public void setTexte(String s) {etiquette.setText(s);}
public String getTexte() {return etiquette.getText();}
}
```


Application à l'exemple (2)

- Composant Bouton

  Rappel : le composant Bouton doit lui aussi avoir deux propriétés, « couleur » et « nom »

```
import java.awt.*;
import java.awt.event.*;
import java.util.*;
public class Bouton extends Panel {
private Button bouton;
public Bouton() {bouton = new Button();add(bouton);}
public void setNom(String s){bouton.setLabel(s);}
public String getNom() {return bouton.getLabel();}
public Color getCouleur() {return bouton.getBackground();}
public void setCouleur(Color c){bouton.setBackground(c);}
}
```


Autre illustration

Une propriété peut être le résultat d'un calcul :

```
public class Eleve {
private int note;
public boolean isBon() {
return note >= 14;
}
}
```

⇒ note = propriété accessible en lecture

Propriétés indicées

- Principe :
  - notion sous−jacente de « tableau » :

    pour différents indices, la propriété a des valeurs différentes
- Méthodes d'accès
  - globales

    public *typePropriété* [] get*NomPropriété*();

    public void set*NomPropriété*(*typePropriété* []);
  - à un indice donné :

    public *typePropriété* get*NomPropriété*(int indice);

    public void set*NomPropriété*(int indice, *typePropriété* val);

Génération d'événements

- Rappel : un bean peut :
  - générer des événements d'un type donné
  - « écouter » des événements générés par un autre bean

- Conséquences :
  - un bean pouvant générer des événements de type *TypeEvt* doit implanter les méthodes :

```
public void addTypeEvtListener(TypeEvtListener el);
public void removeTypeEvtListener(TypeEvtListener el);
```


Gestion des événements

- Principe :
  - lorsqu'un bean génère un événement, celui−ci doit être envoyé à tous les « écouteurs » de cet événement.

- Conséquence :

  le bean générant l'événement doit :
  - gérer la liste des écouteurs
  - propager l'événement à chacun des écouteurs

Application à l'exemple (1)

- Rappel :

  le bouton est susceptible d'envoyer des événements « MouseListener » lorsque l'on clique dessus.
- Première étape : gérer la liste des écouteurs
  1. Conserver la liste par une variable d'instance

```
private Vector listeEcouteursBouton;
```

  2. Initialiser cette liste dans le constructeur :

```
listeEcouteursBouton = new Vector();
```

  3. Ajouter tout nouvel écouteur à cette liste :

```
public void addMouseListener(MouseListener ml) {
    listeEcouteursBouton.addElement(ml);}
```

  4. Supprimer tout écouteur le demandant :

```
public void removeMouseListener(MouseListener ml) {
    listeEcouteursBouton.removeElement(ml);}
```


Application à l'exemple (2)

- Deuxième étape : propager les événements

  Ici, seul le clic sur le bouton nous intéresse
  1. Créer une classe mouchard pour l'événement {

```
class Mouchard extends MouseAdapter {
```

  2. Y redéfinir la méthode correspondant à un clic : cette méthode doit appeler la méthode correspondante de tous les écouteurs

```
public void mouseClicked(MouseEvent me) {
  for (int i = 0; i < listeEcouteursBouton.size(); i++)
  ((MouseListener) listeEcouteursBouton.elementAt(i)).mouseClicked(me);
}}
```

  3. Instancier la classe Mouchard et associer cette instance au bouton dans le constructeur de la classe Bouton

```
bouton.addMouseListener(new Mouchard());
```

Relier des actions à des événements

- Principe

  On ne modifient pas directement les composants, mais on passe par des adaptateurs
- Qu'est−ce−qu'un adaptateur ?
  - Une classe construite spécialement
  - Instancie les méthodes d'écoute d'un événement d'un composant
  - Chacune de ces méthodes appelle l'action à exécuter dans l'autre composant
- Avantages du principe :
  - Les composants restent génériques
  - Procédé automatisable (un des rôle de la beanbox)

Application à l'exemple

- Compléter la classe étiquette

  celle−ci doit en effet contenir les deux actions direOK et direKO :

```
public void direOK() {setTexte(« OK »);}
public void direKO() {setTexte(« KO »);}
```

- Créer les adaptateurs

  deux adaptateurs (un par bouton). Exemple du premier :

```
import java.awt.event.*;
public class adaptateur1 implements MouseListener {
private Etiquette cible;
public void setTarget(Etiquette t) {target = t;}
public void MouseClicked(MouseEvent me) {target.direOK();}
public void mouseEntered(MouseEvent me) {}
public void mouseExited(MouseEvent me) {}
  public void mousePressed(MouseEvent me) {}
  public void mouseReleased(MouseEvent me) {}
}
```


Principe de fonctionnement de l'exemple

- Initialisation
  - L'adaptateur 1 est associé au bouton « Valider » ;
  - on enregistre l'adaptateur 1 comme « écouteur » d'événements « MouseEvent » émis par le bouton « Valider » ;
  - L'adaptateur 2 est associé au bouton « Annuler » ;
  - on enregistre l'adaptateur 2 comme « écouteur » d'événements « MouseEvent » émis par le bouton « Annuler ».
- Fonctionnement
  - L'utilisateur clique sur le bouton « Valider » ;
  - Un événement mouseClicked est envoyé au « mouchard » du bouton « Valider » ;
  - Le mouchard transmet cet événement à l'adaptateur 1, en appelant sa fonction « mouseClicked() » ;
  - La fonction mouseClicked() de l'adaptateur 1 appelle la fonction direOK() de l'étiquette.

Utilisation des Beans

- Archivage dans un fichier .jar qui contient :
  - un (ou des) beans
  - les fichiers associés (images, données, beaninfo, etc.)
  - un fichier «manifest» précisant quels fichiers sont des beans
- Exemple : makefile pour le fichier Bouton.jar :

```
CLASSFILES = Bouton.class
JARFILE = ../jar/Bouton.jar
all: $(JARFILE)
$(JARFILE) : $ (CLASSFILES) $(DATAFILES)
echo « Name: Bouton.class » >> manifest.tmp
echo « Java−Bean: True » >> manifest.tmp
jar cfm $(JARFILE) manifest.tmp *.class
@/bin/rm manifest.tmp
%.class : %.java
export CLASSPATH;CLASSPATH=.;javac $<
clean:
/bin/rm −f *.class
/bin/rm −f $(JARFILE)
```

Persistance

- Définition de « Composant persistant »
  Composant qui conserve son état entre deux exécutions du programme l'utilisant
- Pourquoi des composants persistants ?
  - Utilisation d'un composant =
    instanciation avec des valeurs de propriétés adaptées
  - Propriétés dépendent de l'état (volatile) de l'instance
- Conséquence
  - Tout Bean doit, *a priori*, implanter la persistance

Implanter la persistance en Java

- Hypothèses :
  - sauvegarder un objet = sauvegarder chacune de ses variables d'instance
  - Java sait sauvegarder les types et classes de base
- Conclusion : ce qu'il faut faire
  - Pour chaque bean :
    - importer le `package java.io.Serializable`;
    - implanter l'interface `Serializable`;
    - disposer d'un constructeur sans argument
  - Pour chaque adaptateur : faire de même

Personnaliser la persistance (1)

- Données non persistantes
  - exemple : descripteur de fichier
  - problème : empêcher la persistance
  - solution : rajouter le mot clef « `transient` » dans la déclaration de la variable
- Variables de classe
  - par défaut, ne sont pas archivées

Personnaliser la persistance (2)

- Redéfinir l'archivage des données :

```
private void writeObject(java.io.ObjectOutputStream out)
```

  - D'abord, appeler la méthode par défaut (recommandé) :
    `defaultWriteObject()`
  - Archiver les données :
    - si type simple : `write`*`Type`*`(valeur)`
    - si classe : `writeObject(objet)`
- Redéfinir la récupération des données :

```
private void readObject(java.io.ObjectInputStream in)
```

  - D'abord, appeler la méthode par défaut (recommandé)
    `defaultReadObject()`
  - Récupérer les données :
    - si type simple : `read`*`Type`*`()`
    - si classe : `readObject()`

Personnaliser la persistance (3)

- Limite de `writeObject()` et `readObject()` :
  - format de stockage fixé par Java
- Pallier ces limites :
  - implanter l'interface `Externalizable`
  - redéfinir les méthodes :
    - `readExternal()`
    - `writeExternal()`

Réflexion et Introspection

- Définitions
  - Réflexion :
    mécanisme permettant de découvrir automatiquement certaines composantes d'une classe (par exemple, ses méthodes).
  - Introspection (en Java) :
    mécanisme permettant à une classe de mettre en place explicitement un descriptif de certaines de ses composantes.
- Avantage de chacune des techniques
  - Réflexion : aucun travail n'est demandé au développeur
  - Introspection : Les informations sont plus ciblées
- En Java
  - Réflexion : passer par le package `java.lang.reflect`
  - Introspection : pour les beans, déclarer une nouvelle classe

Réflexion (1)

- Etape préliminaire :
  - créer un objet de type `java.lang.Class` pour la classe sur laquelle on veut des informations
  - moyen :
    - si la classe existe à la compilation :
      - soit à partir d'une instance existante :
        ```
        Class c = monObjet.getClass();
        ```
      - soit à partir d'une instance d'une sous−classe :
        ```
        Class temp = monObjet.getClass();
        Class c = temp.getSuperclass();
        ```
      - soit en créant une instance
      - soit à partir de son nom (voir plus bas)
    - Sinon :
      - si son nom est connu :
        ```
        Class c = MaClasse.class;
        ```
      - sinon, si son nom est dans une variable s de type String :
        ```
        Class c = Class.forName(s);
        ```


Réflexion (2)

On suppose que c désigne un objet de type java.lang.Class

- Connaître le nom d'une classe : `String nom = c.getName();`
- Connaître les modificateurs d'une classe (`public`, `abstract`, `final`) :
  `int m = c.getModifiers();`
  Tester avec :
  - `Modifiers.isPublic(m)`
  - `Modifiers.isAbstract(m)`
  - `Modifiers.isFinal(m)`
- Connaître la superclasse : `Class sup = c.getSuperclass();`
- Connaître les interfaces implantées par une classe :
  `Class [] interfaces = c.getInterfaces();`
- Connaître les champs d 'une classe : `Field [] champs = c.getFields();`
- Connaître les méthodes d 'une classe : `Method [] methodes = c.getMethods();`
- Connaître les constructeurs d'une classe :
  `Constructor [] constructeurs = c.getConstructors();`

Réflexion (3) : Informations sur un champ

Soit f une variable de type Field

- Nom du champ :
```
String nom = f.getName();
```
- Type du champ :
```
Class c = f.getType();
String type = c.getName();
```
- Connaître la valeur du champ pour un objet donné :
```
valeur = f.get(objet);
```
- Modifier la valeur du champ pour un objet donné :
```
f.set(objet, valeur);
```


Réflexion (4) Informations sur une méthode

Soit m une variable de type Method

- Nom de la méthode :
```
String s = m.getName();
```
- Classe de déclaration de la méthode :
```
Class c = m.getDeclaringClass();
```
- Type de retour de la méthode :
```
Class c = m.getReturnType();
```
- Types des paramètres de la méthode :
```
Class [] types = m.getParameterTypes();
```
- Appeler la méthode sur un objet avec certains paramètres :
```
Object retour = m.invoke(objet, paramètres)
```
(paramètres est un tableau d'objets)

Introspection (1) : introduction

- Principe
  Pour un bean appelé MonBean, définir une classe MonBeanBeanInfo qui :
  - implante l'interface BeanInfo
  - est archivée dans le fichier .jar contenant le bean
- Utilisation
  – hériter de la classe SimpleBeanInfo qui définit (toutes les méthodes de l 'interface BeanInfo (mais les définit de façon vierge
  – Créer une constante de type Class contenant la classe du composant auquel la classe BeanInfo est rattachée

Introspection (2) : L'interface BeanInfo

- Quelques méthodes :
  – Liste des propriétés :
    PropertyDescriptor [] getPropertyDescriptors()
  – Liste des méthodes :
    MethodDescriptor [] getMethodDescriptors()
  – Liste des événements pouvant être générés :
    EventSetDescriptor [] getEventSetDescriptors()
- Convention
  – renvoyer `null` si le mécanisme de réflexion doit être utilisé

Introspection (3)
Spécifier les propriétés du composant

- Préliminaire : définir une constante sur la classe à décrire :
  `private final static Class beanclass = nomClasse.class`
- Dans la méthode getPropertyDescriptors() :
  - Pour chaque propriété :
    - créer un objet de type PropertyDescriptor :
      - si les méthodes « get » et « set » sont standards :
        `new PropertyDescriptor (nomPropriété, beanClass)`
      - sinon, au choix :
        `new PropertyDescriptor (String nomPropriété, beanClass, String nomMethodeGet, String nomMethodeSet)`
        `new PropertyDescriptor (String nomPropriété, beanClass, Method methodeGet, Method methodeSet)`
        (passer `null` si l'une des méthodes n'est pas définie)
      - Préciser si la propriété est liée (voir plus loin)
        `p.setBound(true);`
      - Préciser si la propriété est contrainte (voir plus loin)
        `p.setConstrained(true);`
  - Construire un tableau de toutes ces propriétés et le retourner

Propriétés du composant : cas de l'exemple (1)

```
import java.beans.*;
public class EtiquetteBeanInfo extends SimpleBeanInfo {
private final static Class beanClass = Etiquette.class;
public PropertyDescriptor [] getPropertyDescriptors() {
try {
PropertyDescriptor maCouleur =
new PropertyDescriptor(« couleur », beanClass);
PropertyDescriptor monTexte =
new PropertyDescriptor(« texte », beanClass);
maCouleur.setBound(false); //facultatif
maCouleur.setConstrained(false);  //facultatif
PropertyDescriptor [] tab = {maCouleur, monTexte};
return tab;
}
catch(IntrospectionException ie) {
throw new Error(ie.toString());
}
}
}
```


Propriété du composant : cas de l'exemple (2)

Différences par rapport à la réflexion :

- les propriétés de la superclasse (Panel) qui ne nous intéressaient pas (foreground, background, name, font) ne sont plus listées comme propriétés de la classe Etiquette
- il est possible de nommer différemment les méthodes « get » et « set »

Introspection (4)
Spécifier les méthodes du composant

Principe similaire à getPropertyDescriptors :

Dans la méthode getMethodDescriptors :

- pour chaque méthode, créer un objet de type PropertyDescriptor
- renvoyer un tableau de tous ces objets

N.B. : pour chaque paramètre d'une méthode, il est possible d'associer un court texte explicatif (voir classes MethodDescriptor, ParameterDescriptor et FeatureDescriptor)

Méthodes du composant : cas de l'exemple

```
Public MethodDescriptor [] getMethodDescriptors() {
try {
MethodDescriptor ok = new MethodDescriptor(beanClass.getMethod(« direOK », null));
MethodDescriptor ko = new MethodDescriptor(beanClass.getMethod(« direKO, null));
MethodDescriptor [] tab = {ok, ko};
return tab;
}
catch (NoSuchMethodException e) {
throw new Error(e.toString());
}
}
```

- Remarques :
  - `getMethod` prend deux paramètres :
    - le nom de la méthode
    - la liste des paramètres de la méthodes (`null` si pas de paramètres)
  - là encore, ceci évite de voir les méthodes de la superclasse

Introspection (5) : Spécifier les événements émis par le composant

- Principe similaire aux propriétés et méthodes, mais on passe par la méthode `EventSetDescriptors()` :
- Construction d'un EventSetDescriptor :
  - constructeur de base :
    `EventSetDescriptor(Class source, String nomEvénement, Class écouteur, String nomMethodeDeTraitement)`
  - valable si :
    - l'événement nomEvénement est de type nomEvénementEvent
    - une seule méthode sert à traiter l 'événement dans la classe « écouteur » associée
    - les méthodes pour se faire connaître comme écouteur s'appellent `add`*NomEvénement*`Listener` et `remove`*NomEvénement*`Listener`
  - pour les autres constructeurs, se référer au descriptif de la classe EventSetDescriptor (ou à l'exemple qui suit)

Evénements émis par le composant : cas de l'exemple

On se place dans la classe BoutonBeanInfo, la classe Etiquette n'ayant aucun événement à émettre

```
public EventSetDescriptor [] getEventSetDescriptors() {
try {
String [] tabnoms = {« mouseClicked », « mouseExited »,
« mouseEntered », « mousePressed »,
« mouseReleased »};
EventSetDescriptor esd = new EventSetDescriptor(beanClass,
« mouse », MouseListener.class, tabnoms,
« addMouseListener », « removeMouseListener »);
EventSetDescriptor [] tab = {esd};
return tab;
}
catch (IntrospectionException ie) {
throw new Error(ie.toString());
}
}
```


Introspection (6) : Autre fonction− nalités liées à l'interface BeanInfo

- Méthode `getAdditionnalBeanInfo()` :
  – permet de récupérer des informations provenant d'une autre classe implantant l'interface BeanInfo, notamment une classe mère
- Méthode `getIcon()` :
  – permet d'associer au composant une icône pour le représenter
- Spécification d'éditeurs adaptés aux propriétés

Propriétés liées

- Définition

  Une propriété liée est une propriété dont tout changement de valeur est signalé aux objets qui le désirent immédiatement APRES son changement
- Implantation
  - Pour les classes implantant une telle propriété :
    - implanter les deux méthodes :
      - `public void addPropertyChangeListener(PropertyChangeListener l);`
      - `public void removePropertyChangListener(PropertyChangeListener l);`
    - prévenir tous les objets « à l'écoute » dès que la valeur de la propriété est changée
    - N.B. : la classe PropertyChangeSupport aide grandement
  - Pour les classes « à l'écoute » :
    - implanter une méthode traitant les événements PropertyChangeEvent
    - s'enregistrer comme « écouteur » auprès de l'objet détenteur de la propriété
  - Remarque : il est possible d'avoir un contrôle plus fin, par propriété, en utilisant les méthodes :
    - `addPropertyChangeListener(String nomPropriété, PropertyChangeListener l);`
    - `removePropertyChangeListener(String nomPropriété, PropertyChangeListener l);`

La classe PropertyChangeSupport

- Rôle :
  - gérer une liste d'écouteurs de changement de valeur
  - envoyer à tous les membres de la liste tout événement de changement de valeur d'une propriété les concernant
- Méthodes implantées :
  - `PropertyChangeSupport(Object)`
    le constructeur
  - `addPropertyChangeListener(PropertChangeListener)`
    pour ajouter un écouteur
  - `removePropertyChangeListener(PropertyChangeListener)`
    pour supprimer un écouteur
  - `firePropertyChange(PropertyChangeEvent)`
    pour demander l'envoi d'un événement à tous les écouteurs

Propriétés liées : enrichissement de l'exemple (1)

- Modification désirée

  On souhaite que la classe Etiquette signale tout changement de son texte
- Implantation
  - Ajouts :

```
import java.beans.*;
private PropertyChangeSupport gestionnaireChangements = new
    PropertyChangeSupport( this );
public void addPropertyChangeListener(PropertyChangeListener l) {
gestionnaireChangements.addPropertyChangeListener(l);}
public void removePropertyChangeLister(PropertyChangeListener l) {
gestionnaireChangements.removePropertyChangeListener(l);}
```

  - Tout changement du texte doit être envoyé à tous les écouteurs. La méthode setTexte devient :

```
public void setTexte(String s) {
    String ancien = etiquette.getText();
    etiquette.setText(s);
    gestionnaireChangements.firePropertyChange(« texte », ancien, s);}
```

  - Modifier la classe `EtiquetteBeanInfo` pour déclarée la propriété comme liée dans la méthode `getPropertyDescriptors` :

```
monTexte.setBound( true );
```

  - N.B. : si le type de la propriété est un type de base, avant d'appeler firePropertyChange, il faut encapsuler les valeurs de la propriété dans un objet de la classe adéquat du package java.lang.

Propriétés liées : enrichissement de l'exemple (2)

- Modification désirée

  On souhaite que le bouton change de couleur lorsqu'un événement survient. La classe bouton doit donc implanter une méthode susceptible de recevoir des événements de type `PropertyChangeEvent`.
- Implantation

```
public void changement(PropertyChangeEvent e) {
//e.getPropertyName() pour avoir le nom
//e.getOldValue() pour avoir l'ancienne valeur
//e.getNewValue() pour avoir la nouvelle valeur
if (getCouleur().equals(Color.red)) setCouleur(Color.blue);
else setCouleur(Color.red);
}
```

Propriétés liées : Enrichissement de l'exemple (3)

- Modification de la classe BoutonBeanInfo

```
public MethodDescriptor [] getMethodDescriptors() {
try {
Class param = PropertyChangeEvent.class;
Class [] listeparam = {param};
MethodDescriptor pc = new MethodDescriptor(
beanClass.getMethod(« changement », listeparam));
MethodDecriptor [] tab = {pc};
return tab;
}
catch (NoSuchMethodException e) {
throw new Error(e.toString());
}
}
```


Propriétés liées : Enrichissement de l'exemple (4)

- Modification désirée

  on souhaite que la modification du texte de l'étiquette implique un changement de couleur d'un bouton.

- Solution :

  associer à l'événement PropertyChange de l'étiquette l'action « changement » du bouton, soit par la beanbox, soit en écrivant directement un adaptateur

- Implantation de l'adaptateur :

```
import Bouton;import java.beans.*;
public class AdaptateurChangement implements
  java.beans.PropertyChangeListener {
private Bouton target;
public void setTarget(Bouton t) {target = t;}
public void propertyChange(PropertyChangeEvent evt) {
target.changement(evt);
}}
```


Propriétés contraintes : généralités

- Définition

  Une propriété contrainte est une propriété dont les changements de valeur demandés peuvent être refusés par le détenteur de la propriété ou bien par un (d') autre(s) objet(s) « Veto »

- Principe général

  l'objet détenteur d'une propriété contrainte :
  - précise que la méthode « set » associée peut envoyer une exception ;
  - instancie une méthode permettant à d'autres objets de s'enregistrer comme « veto » ;
  - prévient tous les « vetos » lorsque la méthode « set » est appelée, avant que le changement de valeur ait effectivement lieu ;
  - si le changement de valeur est refusé, il peut être intéressant de prévenir tous les objets vetos.

Propriétés contraintes : application

- pour l'objet détenteur de la propriété :
  - la méthode « set » devient :

```
public void setNomPropriété (typePropriété val) throws
java.beans.PropertyVetoException
```

  - ajout des méthodes :

```
public void addVetoableChangeListener(VetoableChangeListener v);
public void removeVetoableChangeListener(VetoableChangeListener
  v);
```

- pour l'objet « veto »
  - implanter une méthode traitant les événements PropertyChangeEvent et pouvant renvoyer une exception PropertyVetoException
  - s'enregistrer auprès de l'objet détenteur de la propriété comme veto
- N.B. : la classe VetoableChangeSupport peut aider grandement
- Remarque : on peut avoir un contrôle plus fin par propriété au moyen des méthodes :

```
public void addVetoableChangeListener(String nomPropriété,
VetoableChangeListener vcl);
public void removeVetoableChangeListener (String nomPropriété,
VetoableChangeListener vcl);
```

La classe VetoableChangeSupport

- Rôle :
  - gérer une liste de Vetos
  - envoyer à tous les membres de la liste tout événement de changement de valeur d'une propriété les concernant
- Méthodes implantées :
  - `VetoableChangeSupport(Object)`
    le constructeur
  - `addVetoableChangeListener(PropertChangeListener)`
    pour ajouter un veto
  - `removeVetoableChangeListener(PropertyChangeListener)`
    pour supprimer un veto
  - `firePropertyChange(PropertyChangeEvent)`
    pour demander l'envoi d'un événement à tous les vetos

Propriétés contraintes : Enrichissement de l'exemple (1)

- Modification désirée
  Le changement de couleur du bouton doit pouvoir être interdit par d'autres composants
- Implantation dans la classe Bouton :
  - on ajoute :

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

  - on modifie setcouleur() :

```
public void setCouleur(Color c) throws PropertyVetoException {
gestionnaireVeto.fireVetoableChange(getCouleur(),c);
bouton.setBackground(c);}
```

  - on modifie aussi le profil de la méthode changement qui doit maintenant être susceptible de renvoyer une exception PropertyVetoException (elle fait en effet appel à setCouleur)

Propriétés Contraintes : Enrichissement de l'exemple (2)

Implantation dans la classe BoutonBeanInfo

- ajout à la méthode getPropertyDescriptors :

```
maCouleur.setConstrained(true);
```

- nouvelle version de getEventSetDescriptors :

```
public EventSetDescriptors [] getEventSetDescriptors() {
try {
String [] tabnoms = {« mouseClicked », « mouseExited »,
      « mouseEntered », « mouseReleased », « mousePressed »};
EventSetDescriptor esd = new EventSetDescriptor(beanClass,
      « mouse », MouseListener.class, tabnoms,
      « addMouseListener », « removeMouseListener »);
EventSetDescriptor esd2 = new EventSetDescriptor(beanClass,
      « propertyChange », VetoableChangeListener.class,
      « vetoableChange »);
EventSetDescriptor [] tab = {esd, esd2}; return tab;}
catch (IntrospectionException ie) {throw new Error(ie.toString());}}
```


Propriétés Contraintes : Enrichissement de l'exemple (3)

- Modification désirée
  un nouvel objet n'autorisera à passer à la couleur « rouge » que si la couleur précédente était « bleu » ou « rose ».
- Implantation

```
import java.awt.*;
import java.beans.*;
public class Contrôle {
public void valider(PropertyChangeEvent evt) throws PropertyException {
Color ancienne = (Color) evt.getOldValue();
Color nouvelle = (Color) evt.getNewValue();
if (nouvelle.equals(Color.red) && (!ancienne.equals(Color.blue) &&
  !ancienne.equals(Color.pink))
throw new PropertyVetoException(« Interdit », evt);
}
}
```

Propriétés contraintes : Enrichissement de l'exemple (4)

- Modification désirée

  On souhaite associer une instance du composant « Contrôle » à la propriété « couleur » du bouton « Valider ».

- Solutions
  - Au moyen de la beanbox :

    associer la méthode VetoableChange de traitement de l'événement « PropertyChangeEvent » du bouton « Valider » à l 'action « valider » de la classe Contrôle
  - En écrivant soi−même son adaptateur

Personnalisation d'un Bean

Deux possibilités

- éditeur de propriété :

  permet de modifier la valeur d'une seule propriété à laquelle il est rattaché

- Configurateur (customizer) :

  permet de configurer tout le bean auquel il est associé.

Editeur de propriété

- Principe de fonctionnement
  - implante lui−même une propriété liée (c'est elle qui est modifiée directement par l'éditeur)
  - génère un événement pour informer le composant auquel la propriété initiale appartient lorsque la propriété de l'éditeur est changée.
- Types d'éditeurs : 3 types possibles
  - utilisation d'une zone de texte avec fonctions de conversion valeur <−> texte
  - utilisation d'une liste de possibilité avec fonctions de conversion valeur <−> texte
  - définition d'un éditeur *ad hoc*

Utilisation d'une zone de texte

- Définir une propriété de l'éditeur du type de la propriété du composant
- Etendre la classe PropertyEditorSupport
- Implanter les méthodes suivantes :
  - public void setValue(Object o) : pour récupérer l'état courant
  - public Object getValue() : pour modifier l'état courant
  - public String getAsText() : pour convertir la valeur en texte
  - public void setAsText(String s) : pour convertir le texte saisi en une valeur et informer le bean du changement de la valeur de la propriété par un firePropertyChange

Zone de texte : exemple

- But

  définir un éditeur pour la propriété « couleur » de la classe Bouton

- Implantation

```
import java.beans.*;
import java.awt.*;
public class CouleurEditor extends PropertyEditorSupport {
private Color couleur;
public void setValue(Object o) {couleur = (Color) o;}
public Object getValue() {return couleur;}
public String getAsText() {
if (couleur.equals(Color.blue)) return « Bleu »;
else if (couleur.equals(Color.red)) return « Rouge »;
else return « Divers »;}
public void setAsText(String s) {
if (s.equals(« Rouge »)) couleur = Color.red;
else if (s.equals(« Bleu »)) couleur = Color.blue;
else couleur = Color.black;
firePropertyChange();}
}
```


Utilisation d'une zone de choix

- Principe :
  - base : analogue à celui de la zone de texte
  - en plus : rajout d'une méthode String [] getTags() renvoyant la liste des choix possibles
- Exemple :
  - But : n'autoriser que les couleurs bleue, rouge et noire.
  - Implantation : rajouter la méthode :

```
public String [] getTags() {
String [] s = {« Bleu », « Rouge », « Noir »};
return s;
}
```


Définir un éditeur personnalisé

- Affichage de la propriété :
  - préciser que l'on peut afficher la valeur de la propriété dans le rectangle associé : méthode `isPaintable()`
  - définir comment afficher cette valeur : `paintValue()`
- Edition de la propriété :
  - dire que l'on a un éditeur personnalisé : méthode `supportsCustomEditor()`
  - créer cet éditeur au moyen d'une nouvelle classe héritant de « Component »
  - renvoyer une instande de l'éditeur au besoin : `getCustomEditor()`

Editeur personnalisé : Exemple (1)

- But :
  - Mettre en place le choix de la couleur par 3 boutons radios.
  - Affichage : la couleur dans le rectangle, le nom centré en noir (ou blanc si la couleur choisie est le noir)
- Implantation (après suppression de getAsText() et setAsText())

```
public boolean isPaintable() {return true;}
public void paintValue(Graphics g, Rectangle rec) {
// Pour faire les choses proprement
Color ancienne = g.getColor();
g.setColor(couleur);
g.fillRect(rec.x, rec.y, rec.width,rec.height);
if (g.getColor().equals(Color.black)) g.setColor(Color.white);
else g.setColor(Color.black);
FontMetrics fm = g.getFontMetrics();
String s = null;
if (couleur.equals(Color.blue)) s = « Bleu »;
else if (couleur.equals(Color.red)) s = « Rouge »;
else s = « Noir »;
int largeur = fm.stringWidth(s);
g.drawString(s, rec.x+(rec.width-largeur)/2, rec.y+rec.height/2);
g.setColor(ancienne);}
```

Editeur personnalisé : Exemple (1)

- Définition de l ’éditeur dans une classe interne :

```
public boolean supportsCustomEditor() {return true;}
public Component getCustomEditor() {  return new EditeurPerso(this );}
public class EditeurPerso extends Panel implements ItemListener {
// Panel est bien une sous-classe de Component
Checkbox bleu, rouge, noir; CheckboxGroup g;
CouleurEditor editeur;
p      EditeurPerso (CouleurEditor ed) {
editeur = ed; setLayout(ne  GridLayout(3,1));
g = ne  CheckboxGroup();
boolean etatBleu = fa  ; boolean etatRouge = fa  ; boolean etatNoir = fa  ;
if ((Color)ed.getValue()).equals(Color.blue)) etatBleu = tr  ;
el  if  ((Color)ed.getValue()).equals(Color.red)) etatRouge = tr  ;
el  etatNoir = tr
bleu = ne  Checkbox(« Bleu », g, etatBleu); add(bleu);bleu.addItemListener(this);
  rouge = n  Checkbox(« Rouge », g, etatRouge); add(rouge);
rouge.addItemListener(this);
  noir = n  Checkbox(« Noir », g, etatNoir); add(noir); noir.addItemListener(this);
}
public void itemStateChanged(ItemEvent ie) {
if (ie.getSource() == bleu) editeur.setValue(Color.blue);
el  if  ie.getSource() == rouge) editeur.setValue(Color.red);
el    diteur.setValue(Color.black);
editeur.firePropertyChange();
}}
```


Associer un éditeur de propriété à une propriété

- Solutions
  - manuelle : par la classe BeanInfo
  - automatique : en fonction du type de la propriété
- Par la classe BeanInfo
  - Principe :
    dans la méthode getPropertyDescriptors, utiliser setPropertyEditorClass
  - Exemple :

```
PropertyDescriptor [] getPropertyDescriptors() {
...
PropertyDescriptor maCouleur = new PropertyDescriptor(« couleur »,
  beanclass);
maCouleur.setPropertyEditorClass(CouleurEditor.class);
...
```


Association automatique propriété−éditeur

- Si la propriété a pour type une classe particulière MaClasse que nous avons définie et que l’éditeur s’appelle MaClasseEditor, il n’y a rien à faire

- Si l ’éditeur porte un autre nom, par exemple MonEditeur, alors il faut ajouter à la classe MaClasse le bloc d ’initialisation statique suivant :

```
static {
java.beans.PropertyEditorManager.registerEditor(MaClasse.class,
   MonEditeur.class);
}
```


Les configurateurs

- But
  - modifier globalement un composant :
    - modifier des propriétés liées
    - modifier des données n’apparaissant pas comme propriétés
- Mise en œuvre :
  configurateur = classe qui :
  - étend la classe java.awt.Component;
  - implante un constructeur par défaut ;
  - implante l’interface java.beans.Customizer, donc ses méthodes :
    - `void addPropertyChangeListener(PropertyChangeListener l);`
    - `void removePropertyChangeListener(PropertyChangListenr l);`
    - `void setObject(Object o) :`
      o est l’objet à personnaliser. Méthode appelée lorsque l’utilisateur active le configurateur. Elle permet de récupérer les valeurs actuelles du composant et de pouvoir activer ses méthodes lorsqu’une propriété doit être modifiée

Actions d'un configurateur

- dans la méthode setObject :
  - s'initialise
- lorsqu'une propriété doit être changée :
  - appelle la méthode correspondante de l'objet
  - appelle la méthode firePropertyChanged().