La Programmation Par Composants 1

Yassamine Seladji

yassamine.seladji@gmail.com

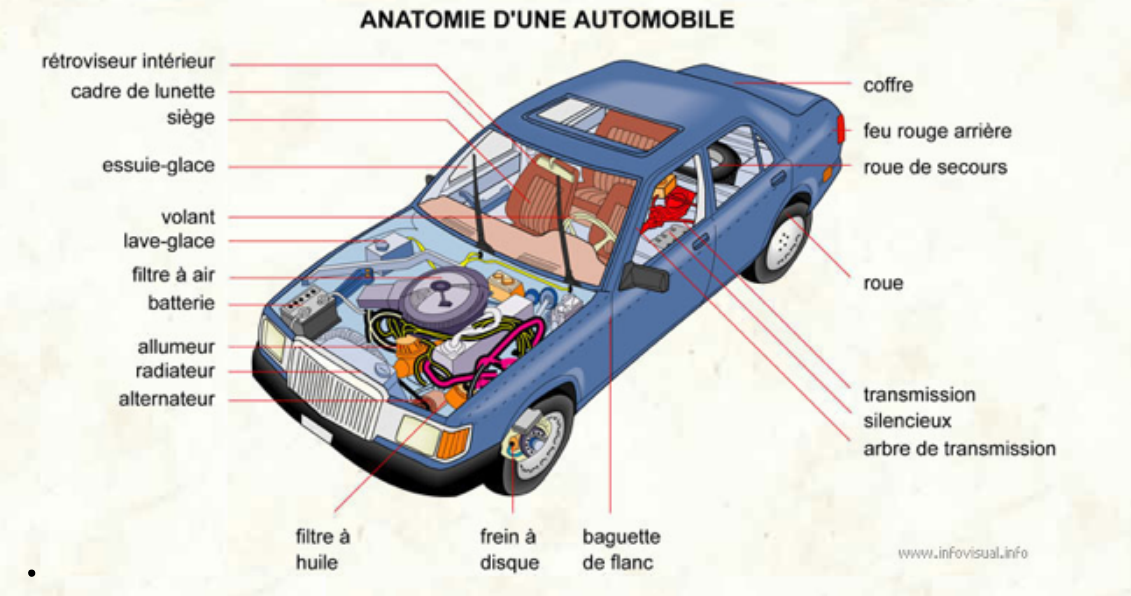
18 janvier 2021

Introduction

On peut résoudre un problème complexe en :

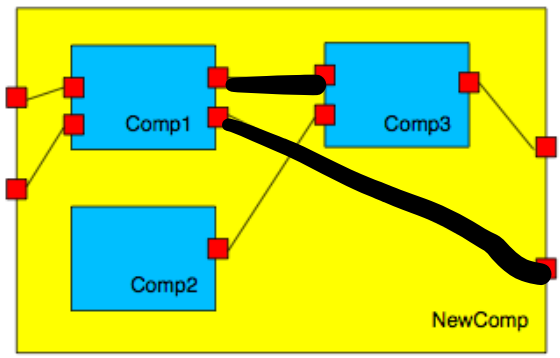
- le décomposant en sous problèmes afin de diminuer sa complexité et de faciliter sa résolution.
- réutilisant ou en adaptant des solutions existantes.

Introduction



Introduction

- Assembler plusieurs composants pour construire un nouveau composant.
- Les composants facilitent la description d'une application complexe.
  - Une famille de composants.
  - Connexion entre ces composants.



La Programmation Orientée Composants

Définition :

La programmation orientée composant :

- est une méthode qui décompose un problème en grandes sections, appelées **composants**
- est utilisé pour le développement de logiciels par agrégation de briques logicielles existantes.

La Programmation Orientée Composants

Définition :

Un composant est une brique logicielle, de taille variable (une classe ... application entière). Un composant doit être caractérisé par :

- sa robustesse : dois définir le comportement voulu, sans bug.

La Programmation Orientée Composants

Définition :

Un composant est une brique logicielle, de taille variable (une classe ... application entière). Un composant doit être caractérisé par :

- sa robustesse : dois définir le comportement voulu, sans bug.
- sa généricité : dois être adaptable à des applications différentes.

La Programmation Orientée Composants

Définition :

Un composant est une brique logicielle, de taille variable (une classe ... application entière). Un composant doit être caractérisé par :

- sa robustesse : dois définir le comportement voulu, sans bug.
- sa généricité : dois être adaptable à des applications différentes.
- son abstraction : dois être utilisable dans des applications différentes (une interface claire).

La Programmation Orientée Composants

Un composant peut être :

- **technique** : ossature du code, éléments d'interface graphique, utilitaire pour manipuler des données...
- **métier** : définit des entités du domaine (client, produit...).
- **applicatif** : utilisé dans le traitement internes d'une application.

Exemple de modèle de programmation orienté composant :

- EJB (Entreprise Java Beans).
- Java beans.
- Le modèle .NET.
- ...

La Programmation Orientée Composants

Plusieurs frameworks et technologies sont basés sur le modèle orienté composants :

- **Bundles** définis dans la plateforme de service OSGi.
- **Component** une plateforme web pour des js et css modulaire.
- **Component Object Model** (OCX/ActiveX/COM) et **DCOM** de Microsoft.
- **Enterprise JavaBeans** de Sun Microsystems (aujourd'hui Oracle).
- ...

JavaBean

Définition

Un **JavaBean** est : ,

- un module autonome pouvant être installé sur différentes plateformes (fichier .jar).
- un modèle de composant proposé par Java.
- une classe Java spéciale.
- une implémentation qui suit les spécifications de l'API **Javabeans**.

JavaBean

Les caractéristiques d'une classe **JavaBean** :

- Son constructeur ne prend pas de paramètres.

JavaBean

Les caractéristiques d'une classe **JavaBean** :

- Son constructeur ne prend pas de paramètres.
- Il doit implémenter l'interface **Serializable**.

JavaBean

Les caractéristiques d'une classe **JavaBean** :

- Son constructeur ne prend pas de paramètres.
- Il doit implémenter l'interface **Serializable**.
- Ses propriétés(attributs) sont toujours privés, et ont des getteurs et setteurs.

JavaBean

Les caractéristiques d'une classe **JavaBean** :

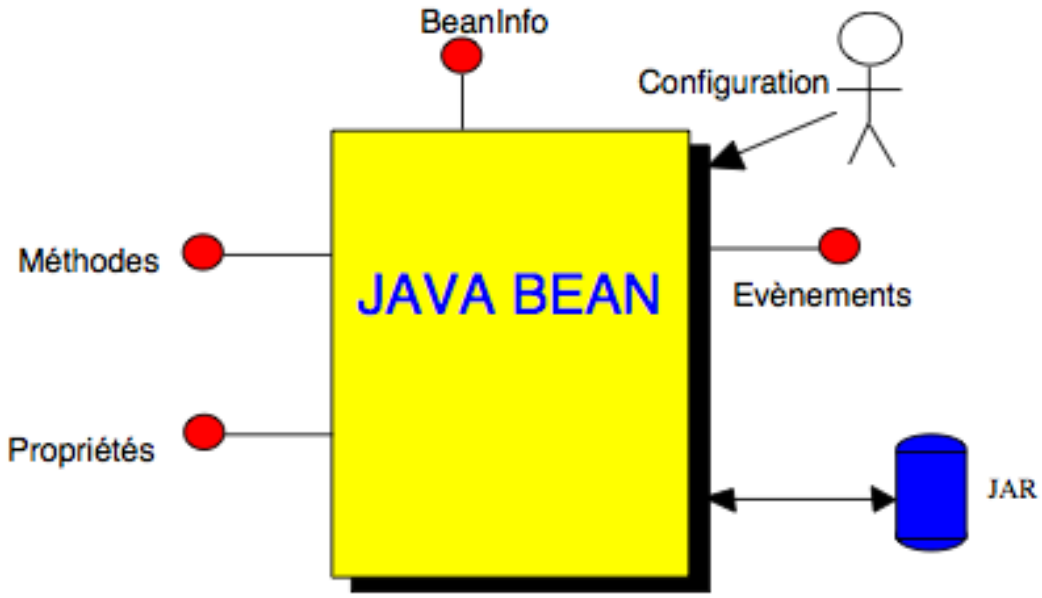
- Son constructeur ne prend pas de paramètres.
- Il doit implémenter l'interface **Serializable**.
- Ses propriétés(attributs) sont toujours privés, et ont des getteurs et setteurs.
- Ses méthodes utilisables par les composants extérieurs doivent être public et gérer les accès concurrents.

JavaBean

Les caractéristiques d'une classe **JavaBean** :

- Son constructeur ne prend pas de paramètres.
- Il doit implémenter l'interface **Serializable**.
- Ses propriétés(attributs) sont toujours privés, et ont des getteurs et setteurs.
- Ses méthodes utilisables par les composants extérieurs doivent être public et gérer les accès concurrents.
- Les beans communiquent via le modèle évènementiel et le pattern d'écouteur (**Listener**).

JavaBean


BeanInfo
Configuration
Méthodes
JAVA BEAN
Evènements
Propriétés
JAR

JavaBean : les propriétés

- Les propriétés sont les attributs de la classe, ils peuvent être de n'importe quel type Java.
- Ils doivent être privées.
- Ils peuvent avoir des accès en lecture, en écriture, en lecture seul ou en écriture seul.

JavaBean : les propriétés

- Les propriétés sont les attributs de la classe, ils peuvent être de n'importe quel type Java.
- Ils doivent être privées.
- Ils peuvent avoir des accès en lecture, en écriture, en lecture seul ou en écriture seul.

Méthodes	Description
get**Attribut**	La méthode permet de récupérer la valeur de **Attribut**.
set**Attribut**	La méthode permet de changer la valeur de **Attribut**.
Is**Attribut**	La méthode retourne un booléen.

JavaBean : les propriétés

```
public class StudentsBean implements java.io.Serializable
{
   private String firstName = null;
   private String lastName = null;
   private int age = 0;

   public StudentsBean() {
   }
   public String getFirstName(){
      return firstName;
   }
   public String getLastName(){
      return lastName;
   }
   public int getAge(){
      return age;
   }
   public void setFirstName(String firstName){
      this.firstName = firstName;
   }
   public void setLastName(String lastName){
      this.lastName = lastName;
   }
   public void setAge(Integer age){
      this.age = age;
   }
}
```

JavaBean : les propriétés liées

- La propriété liée est une propriété qui appartient à un JavaBean et qui est observée par un autre JavaBean. Le mécanisme utilisé implémente le pattern **Observer**.

JavaBean : les propriétés liées

- La propriété liée est une propriété qui appartient à un JavaBean et qui est observée par un autre JavaBean. Le mécanisme utilisé implémente le pattern **Observer**.
- Le JavaBean **observé** intègre un objet de type *PropertyChangeSupport* :

JavaBean : les propriétés liées

- La propriété liée est une propriété qui appartient à un JavaBean et qui est observée par un autre JavaBean. Le mécanisme utilisé implémente le pattern **Observer**.
- Le JavaBean **observé** intègre un objet de type *PropertyChangeSupport* :
  - *PropertyChangeSupport* manipule un ensemble d'écouteurs de type *PropertyChangeListener* qui ont le rôle d'observateurs.

JavaBean : les propriétés liées

- La propriété liée est une propriété qui appartient à un JavaBean et qui est observée par un autre JavaBean. Le mécanisme utilisé implémente le pattern **Observer**.
- Le JavaBean **observé** intègre un objet de type *PropertyChangeSupport* :
  - *PropertyChangeSupport* manipule un ensemble d'écouteurs de type *PropertyChangeListener* qui ont le rôle d'observateurs.
  - *PropertyChangeSupport* permet de notifier aux écouteurs les changements liés à une propriété liée.

JavaBean : les propriétés liées

```
public class StudentBean implements java.io.Serializable {

   private String firstName = null;
   private String lastName =null;
   private PropertyChangeSupport pcs;

   public StudentBean (){
        ...
        pcs = new PropertyChangeSupport();
   }

   public void addPropertyChangeListener(PropertyChangeListener pcl){
        pcs.addPropertyChangeListener(pcl);
   }

   public void removePropertyChangeListener(PropertyChangeListener pcl){
        pcs.removePropertyChangeListener(pcl);
   }

   public void setFirstName(String fn){
      String  temp  = firstName; this.firstName = fn;
        pcs.firePropertyChange("firstName", new String(temp), new String(fn));
   }

   ...

}
```

JavaBean : les propriétés liées

- Le JavaBean **observateur** implémente l'interface *PropertyChangeListener* :
  - Contient la méthode *void propertyChange(PropertyChangeEvent evt)*.
  - la classe *PropertyChangeEvent* encapsule les données qui décrivent le changement de valeur de la propriété.

```
public class StudentScience implements PropertyChangeListener{

   private University univ;

   public StudentScience(){...}

   public void propertyChange(PropertyChangeEvent pce){
       if(pce.getPropertyName().equals("firstName"));

            // Changer la liste des groupes
   }


}
```

JavaBean : les méthodes

- Les classes JavaBeans peuvent aussi avoir des méthodes :
  - ces méthodes doivent être publiques.
  - c'est toute méthode qui ne fait pas partie de la définition d'une propriété.

JavaBean : les évènements

- En plus des propriétés et des méthodes, les beans communiquent par le biais d'évènements.

JavaBean : les évènements

- En plus des propriétés et des méthodes, les beans communiquent par le biais d'évènements.
- Les évènements suivent un pattern de nommage spécifique.

  - Evènement :
    ```
    class NomEvent
    ```
  - Auditeur :
    ```
    interface NomListener
    ```
  - Classe source d'évènements :
    ```
    public void addNomListener(NomListener l)
    public void removeNomListener(NomListener l)
    ```

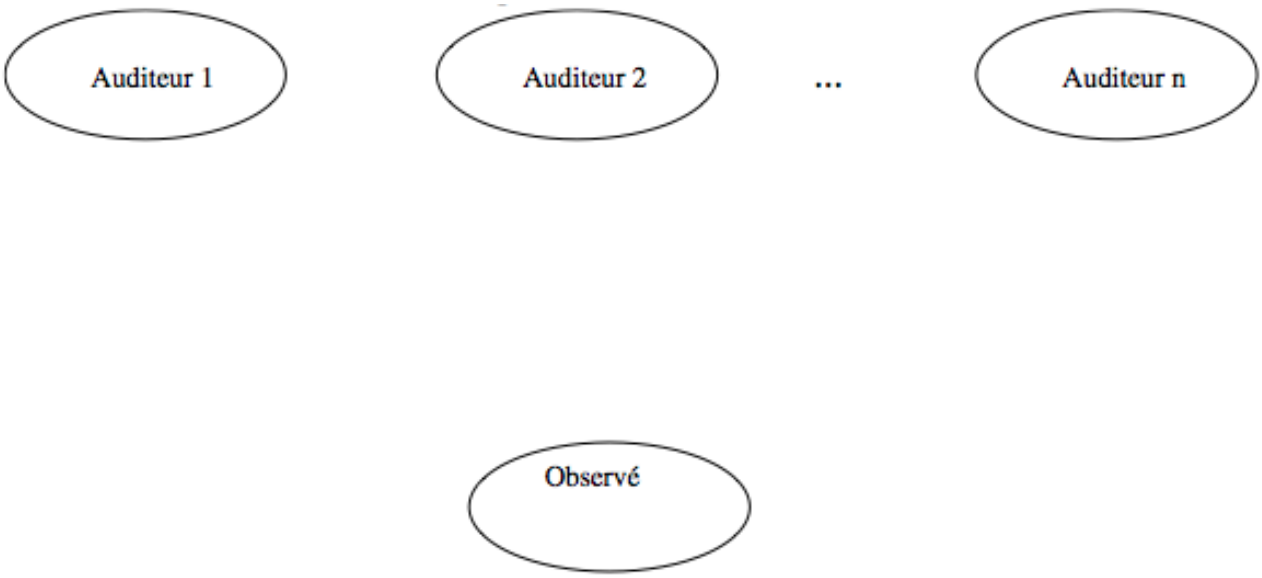
JavaBean : les évènements

- En plus des propriétés et des méthodes, les beans communiquent par le biais d'évènements.
- Les évènements suivent un pattern de nommage spécifique.
  - Evènement :
    ```
    class NomEvent
    ```
  - Auditeur :
    ```
    interface NomListener
    ```
  - Classe source d'évènements :
    ```
    public void addNomListener(NomListener l)
    public void removeNomListener(NomListener l)
    ```
- le type listener doit doit de la classe **java.util.EventListener**.

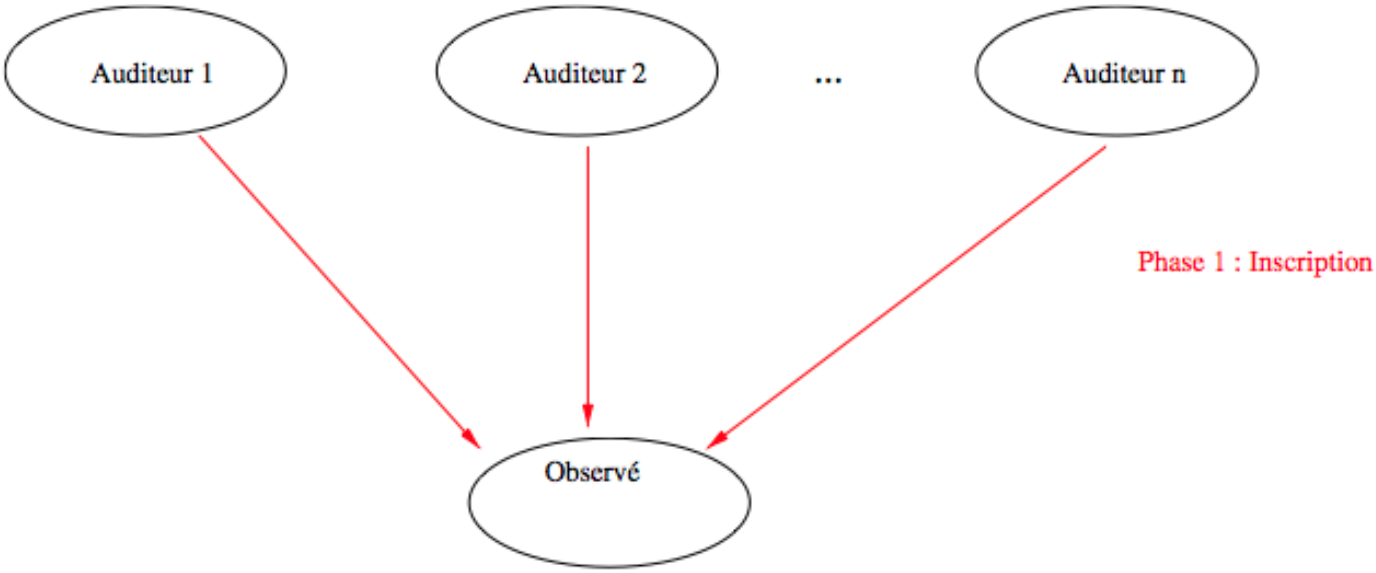
JavaBean : les évènements

- En plus des propriétés et des méthodes, les beans communiquent par le biais d'évènements.
- Les évènements suivent un pattern de nommage spécifique.
  - Evènement :
    ```
    class NomEvent
    ```
  - Auditeur :
    ```
    interface NomListener
    ```
  - Classe source d'évènements :
    ```
    public void addNomListener(NomListener l)
    public void removeNomListener(NomListener l)
    ```
- le type listener doit doit de la classe **java.util.EventListener**.
- Les évènements sont basés su le design pattern **Observer**.

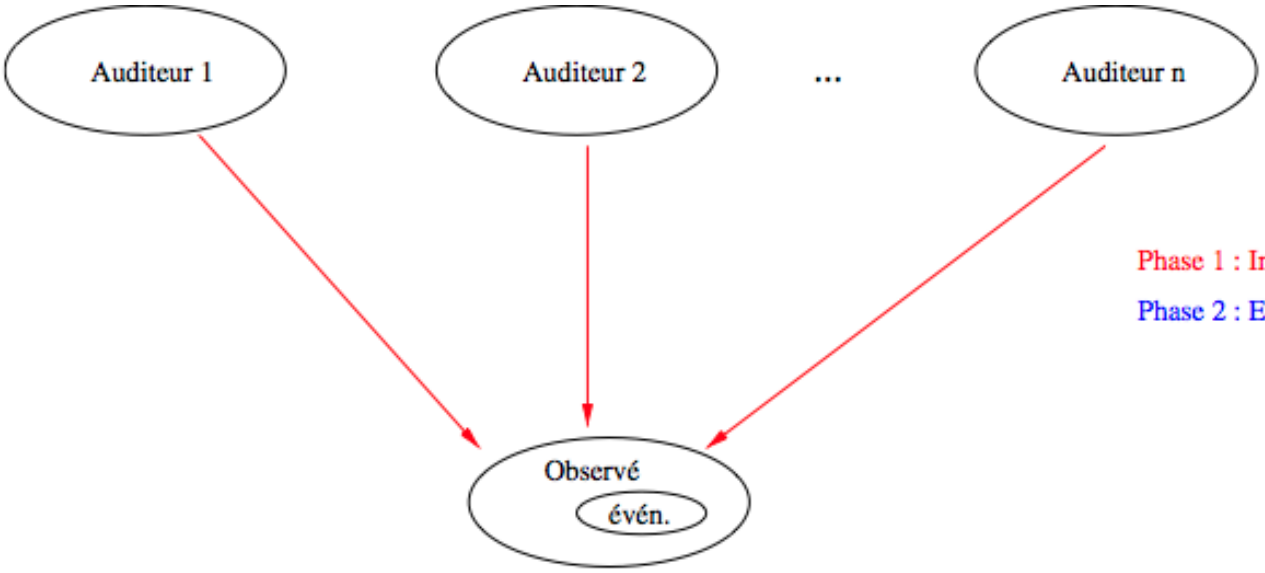
JavaBean : les évènements



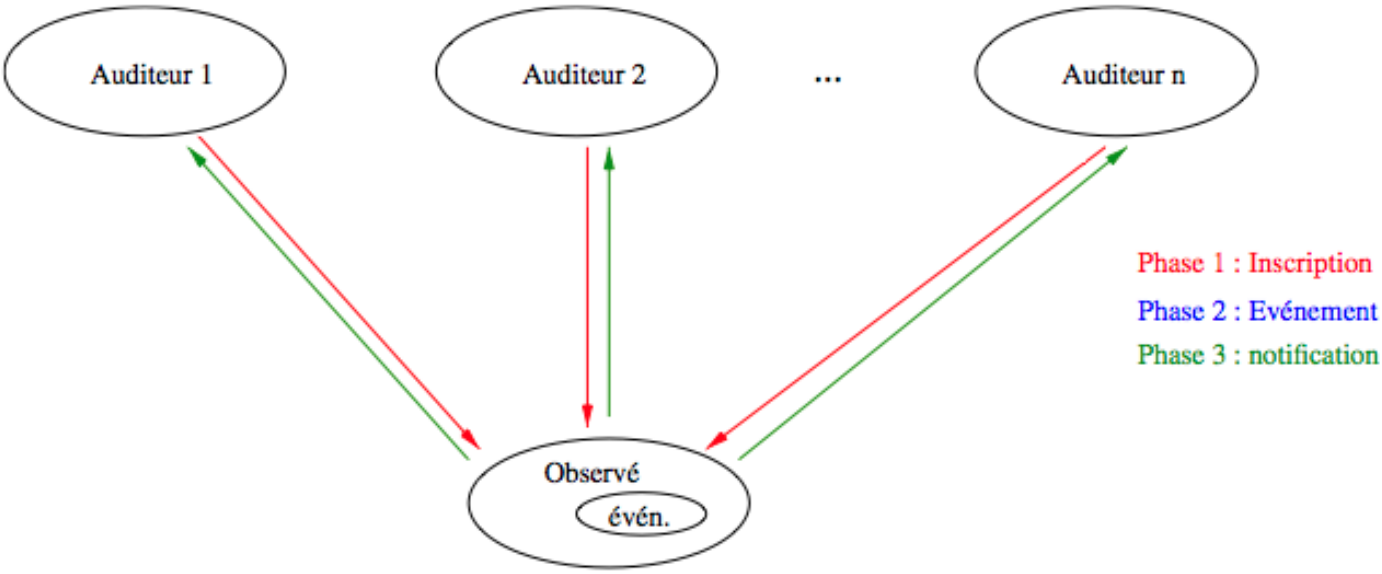
JavaBean : les évènements



JavaBean : les évènements

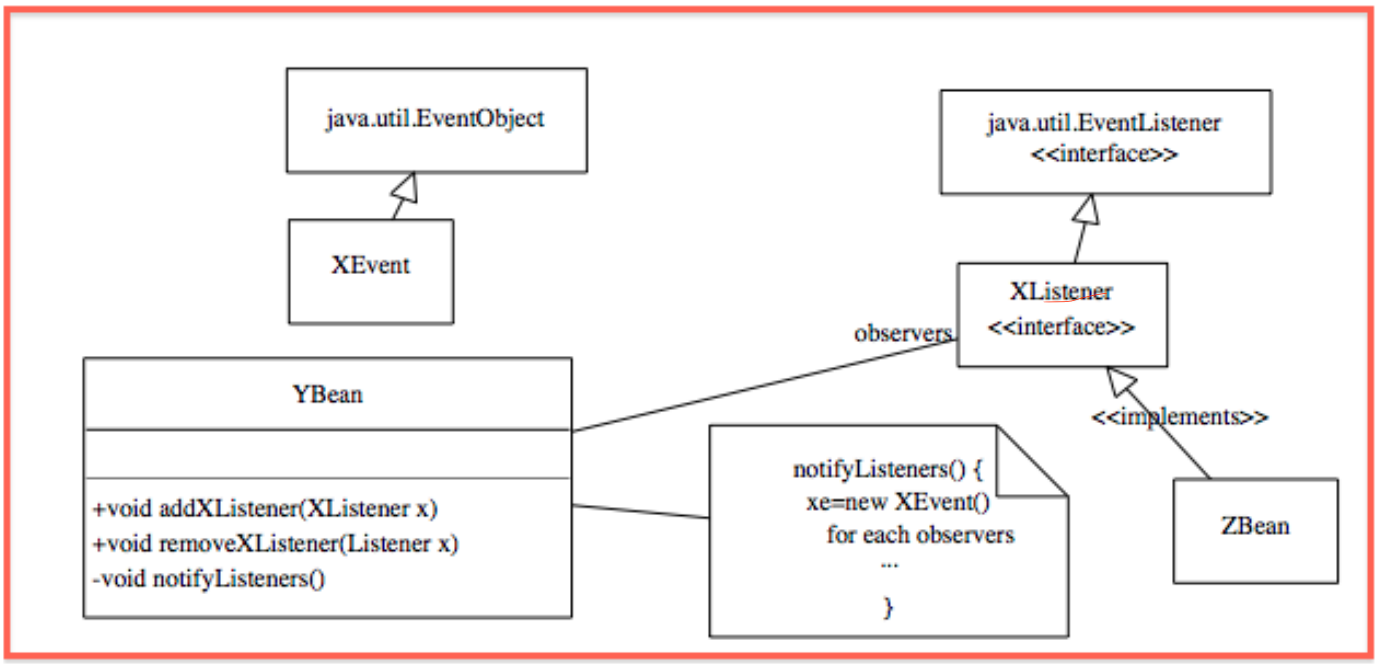


JavaBean : les évènements



JavaBean : les évènements

Exemple :



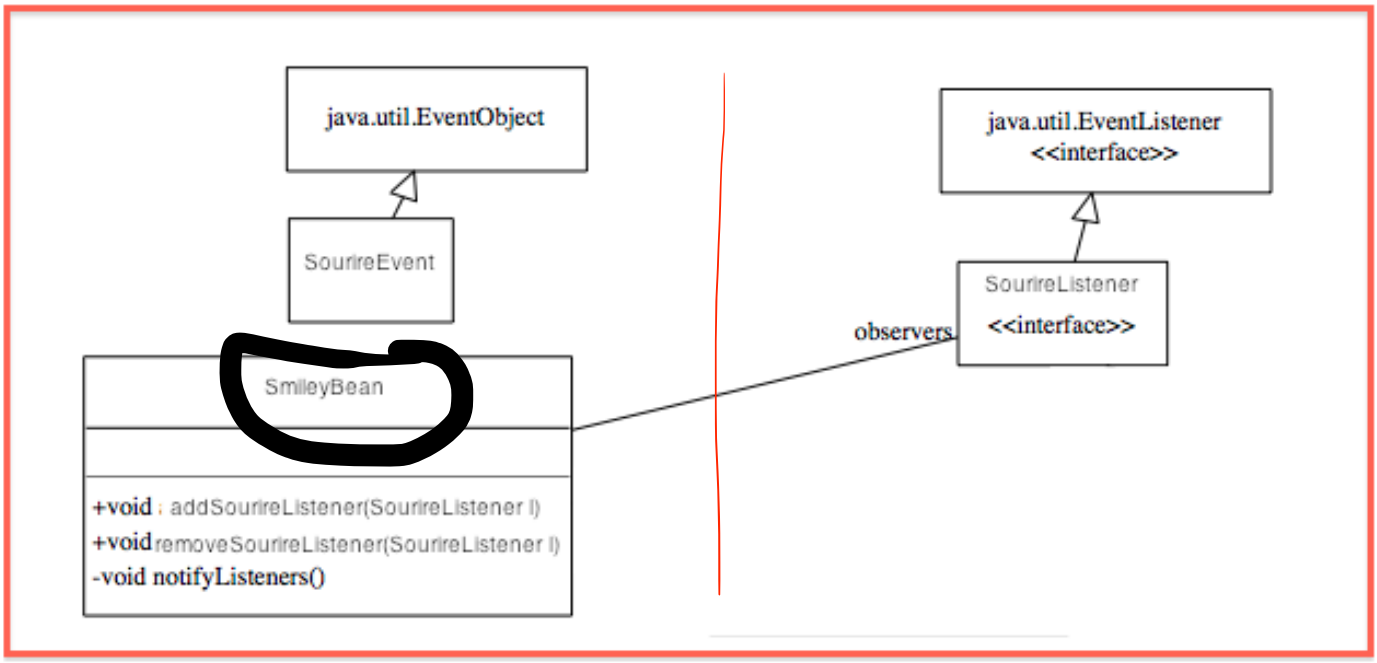
- X $\longrightarrow$ Action.
- Y $\longrightarrow$ Bouton.

JavaBean : les évènements

Exemple :

- Créer un évènement quand un Smiley souris.
- Plusieurs auditeurs peuvent être à l'écoute de cet évènement (implémenter **SourireListener**).
- Créer le bean qui :
  - ajoute et retire les auditeurs (listeners).
  - crée l'évènement **SourireEvent** quand le smiley sourit.
  - notifie l'évènement à tous les auditeurs.

JavaBean : les évènements



JavaBean : les évènements

```
import java.util.EventObject ;

public class SourireEvent extends EventObject {
  public SourireEvent(SmileyBean src) { super(src); }
}
```

```
import java.util.EventListener ;

public interface SourireListener extends EventListener {
  public void devientDrole(SourireEvent e) ;
}
```

JavaBean : les évènements

```
import java.awt.*;
import java.beans.*;
import java.util.ArrayList ;
public class SmileyBean extends Canvas {
  // Private data fields:
  private Color ourColor = Color.yellow;
  private boolean smile = true;
  private ArrayList<SourireListener> listeners ;
  public SmileyBean() {
      this.setSize(250,250);
      this.listeners = new ArrayList<SourireListener>() ;
  }

synchronized
public void addSourireListener(SourireListener l) {
  listeners.add(l) ;
}

synchronized
public void removeSourireListener(SourireListener l) {
  listeners.remove(l) ;
}
```

JavaBean : les évènements

```
  private void notifyListeners() {
    SourireEvent se=new SourireEvent(this) ;
    ArrayList lv =null ;
    // realisation copie (acces concurrent)
    // ex: cas ou un addListener en action
    synchronized(this) {
      lv=(ArrayList)listeners.clone() ;
    }
    for (int i=0;i<lv.size();i++)
      ((SourireListener) lv.get(i)).devientDrole(se) ;
  }
  public void paint(Graphics g) { ... }
}
```

JavaBean : Exercice 1

- Écrire un JavaBean qui représente un abonné.
- Un abonné est caractérisé par :
  - nom et prénom.
  - adresse mail.
  - son centre d'intéret : économie, politique, sport, ...

JavaBean : Exercice 2

- Écrire un JavaBean qui représente une liste d'abonnés.
- Le JavaBean contient :
  - une méthode **ajoutAbonne** : qui permet d'ajouter un abonné.
  - une méthode **supprimeAbonne** : qui permet de supprimer un abonné.