JAVABEANS

1 Généralités

- **Un JavaBean est un composant Java jouant le rôle de brique de base logicielle.**
- **L'architecture d'un JavaBean** repose sur :
    - **des propriétés** (persistantes ou non)
    - **des événements** (modèle événementiel Java 1.1), ses méthodes sont activées par des événements.
    - **des méthodes de traitements des événements**
    - le mécanisme d'**introspection** (reflection)
- Le package **java.beans** leur est dédié.
- Les Beans peuvent être **visibles**, tels les composants graphiques, ou **invisibles**, tels les queues, piles et autres structures de données.
- La **persistance** d'un JavaBean est assurée par la **sérialisation** (sauvegarde de son état) et par la **désérialisation** (restitution de son état lors de son chargement dans une application)
- Un JavaBean ne peut être sérialisé que s'il implémente l'interface de marquage ***java.io.Serializable*** *(interface vide).*

2 Evénements

Le modèle d'évènement de Java, basé sur le principe de la délégation, est utilisé pour les Beans :

- des **classes événements** sous-classes de ***java.util.EventObject***
- des **interfaces de listeners** d'événements (auditeurs, récepteurs), sous classes de ***java.util.EventListener***
- des **classes pour les sources**(émetteurs) d'événements
- Une mise en œuvre de ce modèle est fournie par les composants graphiques du package ***javax.swing***.

2.1 java.util.EventObject

- La classe ***java.util.EventObject*** est la base de tous les événements.

```
public class java.util.EventObject
    extends Object implements java.io.Serializable {
        public java.util.EventObject (Object source) { ... };
        public Object getSource() { ... };
        @Override
        public String toString() { ... };
}
```

- Les sous-classes de la classe ***EventObject*** permettent de définir un type d'événement bien spécifique.

Exemple :

- L'événement **HireEvent** modélise la date d'embauche d'un employé.

```
public class HireEvent extends EventObject {
        private long hireDate;

        public HireEvent (Object source) {
                super (source);
                hireDate = System.currentTimeMillis();
         }

        public HireEvent (Object source, long hired) {
                super (source);
                hireDate = hired;
        }

        public long getHireDate () {
                return hireDate;
        }
}
```

2.2 java.util.EventListener

L'interface ***java.util.EventListener*** est vide, elle sert de marqueur d'interface et doit être sous classée par une interface spécifique.

- Un auditeur d'événements doit implémenter cette interface.
- Un auditeur d'événement désire recevoir une notification quand un événement arrive.
- L'auditeur doit implémenter une méthode de l'interface ayant la sous-classe de l'événement spécifique en paramètre.

Exemple :

- Un auditeur pour l'événement *"date d'embauche d'un employé"* est modélisé par l'interface **HireListener**
- La méthode de l'interface ne respecte pas de **Design Pattern** particulier. Elle doit néanmoins décrire l'événement qui se produit.

```
public interface HireListener extends java.util.EventListener {
        public abstract void hired (HireEvent e);
}
```

2.3 Source de l'évènement

- Un source d'événement définit quand et où un événement arrive.
- Les auditeurs intéressés par l'événement s'enregistrent auprès de la source et recoivent une **notification** lorsque l'événement arrive.

Les méthodes d'enregistrement suivent le Design Pattern :

```
public synchronized void addListenerType(ListenerType l);
public synchronized void removeListenerType(ListenerType l);
```

Evénements et *threads* multiples

- Un auditeur peut recevoir des notifications d'événements faites par des méthodes appelées dans plusieurs *threads* différents.
- Il faut donc prendre des mesures, pour éviter aussi bien des comportements incohérents que des blocages du programme (**étreinte mortelle**).

Des principes à observer :

- il est recommandé que les **méthodes de notification**, dans les auditeurs d'événements, soient **synchronisées**, de manière à ce que les éventuelles réactions à un **même événement**, notifié depuis **plusieurs sources**, prennent place les unes après les autres.

- **l'objet source d'un événement ne doit pas être verrouillé lorsqu'il appelle une méthode de notification sur un auditeur de cet événement** car la réaction de l'auditeur pourrait être l'appel d'une méthode de l'objet source (c'est un cas fréquent) et le **blocage** serait alors probable.
- Pour notifier un événement, l'objet source doit parcourir la liste des auditeurs ; or, la réaction d'un auditeur peut être de modifier cette liste (par exemple, suite à l'événement, l'auditeur s'enlève de la liste, ou bien lui ajoute un nouvel auditeur); il y a alors un risque que le **parcours devienne incohérent.**
- En programmant l'opération *"notifier un événement à tous les auditeurs"* il faut donc **prendre soin de parcourir non pas la liste des auditeurs, mais un clone de cette liste**, produit à cette occasion.

Rappel :

- ArrayList<T> et LinkedList<T> ne sont pas synchronisées.

Cas particulier :

- Pour un auditeur unique (envoi d'événement **unicast**), il faut utiliser la méthode :

```
public synchronized void addListenerType(ListenerType l)
                    throws java.util.TooManyListenersException
```

- Un container de listeners est alors inutile, seule une référence suffit.

Exemple de classe source d'évènement :

La source pour l'événement *"date d'embauche d'un employé"* est modélisée par la classe **Hire.**

```
public class Hire {
    private List<HireListener> hireListeners =
                                new ArrayList<HireListener>();
    ...
    public synchronized void addHireListener(HireListener l) {
            hireListeners.add(l); }
    public synchronized void removeHireListener(HireListener l) {
            hireListeners.remove(l); }

    // Notification de l'événement "date d'embauche d'un employé"
    public void notifyHired () {
        List<HireListener> hireListenersClone;
        // création de l'événement
        HireEvent hireEvent = new HireEvent (this);

         // copie superficielle du container de listeners, seules
        // les références sont copiées par les objets
        // le container copié sera invariant pendant l'envoi de
        // l'événement
        synchronized (this) {
            hireListenersClone =
            List<HireListener>)hireListeners.clone();
        }
```

```
        // activation de la méthode pour les auditeurs
        HireListener hireListener = null;
        for (int i = 0; i < l.size(); i++){
            hireListener = l.get(i);
            hireListener.hired(hireEvent);
        }
}
```

Remarque :

- Pour avoir une ArrayList synchronisée

  List list = Collections.synchronizedList(new ArrayList(...));

3 Propriétés

- Les propriétés définissent les caractéristiques d'un JavaBean.
- Une propriété est un attribut "public", au sens commun du terme, implémentée le plus souvent par une variable d'instance non publique!
- Les méthodes d'accès sont définies par le **schème de conception** (**design pattern**) suivant :
    - accesseur en lecture :

      public void setPropertyName(PropertyType value);
    - accesseur en écriture :

      public PropertyType getPropertyName();

      où ***PropertyName*** est le nom de la propriété et ***PropertyType*** son type de données.
- Si une seule des 2 méthodes est définie, la propriété est en lecture seule (read-only pour get) et en écriture seule (write-only pour set).
- Les propriétés sont de 5 types :
    - Simple
    - Booléenne
    - Indicée
    - Liée
    - Contrainte

3.1 Propriétés simples

- Par convention (design pattern), le nom de la paire de méthodes set/get définit une propriété, une variable d'instance de même nom implémente la propriété.

Exemple :

- Un employé a la propriété "salaire".

```
public class Employee
{
        private int salary;

        public void setSalary (int salary){
                this.salary = salary;
        }

        public int getSalary (){
                return salary;
        }
}
```

3.2 Propriétés booléennes

- Elles utilisent, au lieu de get la méthode :

  public boolean isPropertyName()

Exemple :

```
boolean trained;

public void setTrained (boolean trained){
  this.trained = trained;
}

public boolean isTrained (){
  return trained;
}
```

3.3 Propriétés indicées

- Une propriété indicée est utilisé quand une propriété simple peut prendre une valeur dans un **tableau de valeurs**.
- Le design pattern est :

  public PropertyType[] getPropertyName ()

  public PropertyType getPropertyName (int position)

  public void setPropertyName (PropertyType[] properties)

  public void setPropertyName (PropertyType element, int position)

3.4 Propriétés liées

- Les classes et l'interface nécessaires à la mise en œuvre de ce mécanisme sont offertes par Java dans le package ***java.beans***:
  - ***PropertyChangeEvent*** pour l'événement
  - ***PropertyChangeSupport*** pour la source
  - ***PropertyChangeListener*** pour l'interface des récepteurs

```
public class PropertyChangeEvent extends java.util.EventObject {
    /**
     * Constructs a new <code>PropertyChangeEvent</code>.
     * @param source  The bean that fired the event.
     * @param propertyName  The programmatic name of the property that was
     * changed.
     * @param oldValue  The old value of the property.
     * @param newValue  The new value of the property.
     */
public PropertyChangeEvent(Object source, String propertyName,
                           Object oldValue, Object newValue)

    /**
     * Get the programmatic name of the property that was changed.
     * @return  The programmatic name of the property that was changed.
     * May be null if multiple properties have changed.
     */
     public String getPropertyName() { … }
```

```
    /**
    * Get the new value for the property, expressed as an Object.
    * @return  The new value for the property, expressed as an Object.
    * May be null if multiple properties have changed.
    */
    public Object getNewValue()  { … }

    /**
    * Gets the old value for the property, expressed as an Object.
    * @return  The old value for the property, expressed as an Object.
    * May be null if multiple properties have changed.
    */
    public Object getOldValue()  { … }
}
```

```
public class PropertyChangeSupport implements java.io.Serializable
  /**
   * Constructs a <code>PropertyChangeSupport</code> object.
   * @param sourceBean  The bean to be given as the source for any events.
   */
  public PropertyChangeSupport(Object sourceBean) { … }

   /**
   * Add a PropertyChangeListener to the listener list.
   * The listener is registered for all properties.
   * @param listener  The PropertyChangeListener to be added
   */
   public synchronized void addPropertyChangeListener(
                                        PropertyChangeListener listener) { … }

  /**
   * Remove a PropertyChangeListener from the listener list.
   * This removes a PropertyChangeListener that was registered
   * for all properties.
   * @param listener  The PropertyChangeListener to be removed
   */
  public synchronized void removePropertyChangeListener(
                                      PropertyChangeListener listener) { … }
```

```
 /**
 * Add a PropertyChangeListener for a specific property.  The listener
 * will be invoked only when a call on firePropertyChange names that
 * specific property.
 * @param propertyName  The name of the property to listen on.
 * @param listener  The PropertyChangeListener to be added
 */
public synchronized void addPropertyChangeListener(
        String propertyName, PropertyChangeListener listener)  { …  }

 /**
 * Remove a PropertyChangeListener for a specific property.
 * @param propertyName  The name of the property that was listened on.
 * @param listener  The PropertyChangeListener to be removed
 */
public synchronized void removePropertyChangeListener(
        String propertyName, PropertyChangeListener listener) { …  }

 /**
 * Report a bound property update to any registered listeners.
 * No event is fired if old and new are equal and non-null.
 * @param propertyName  The programmatic name of the property
 * that was changed.
 * @param oldValue  The old value of the property.
 * @param newValue  The new value of the property.
 */
```

```
    public void firePropertyChange(String propertyName,
                                   Object oldValue, Object newValue) { … }

    public void firePropertyChange(String propertyName,
                                   int oldValue, int newValue) { … }

    public void firePropertyChange(String propertyName,
                                   boolean oldValue, boolean newValue) { … }

  /**
   * Fire an existing PropertyChangeEvent to any registered listeners.
   * No event is fired if the given event's old and new values are equals and
   * non-null.
   * @param evt  The PropertyChangeEvent object.
   */
  public void firePropertyChange(PropertyChangeEvent evt) { … }
}
```

```
public interface PropertyChangeListener extends java.util.EventListener {
  /**
   * This method gets called when a bound property is changed.
   * @param evt A PropertyChangeEvent object describing the event source
   *   and the property that has changed.
   */
  void propertyChange(PropertyChangeEvent evt);
}
```

Exemple :

- Reprenons l'exemple du **JavaBean Employee** et faisons de **salary** une **propriété liée**.
- Ainsi si deux personnes "utilisent" la même instance du JavaBean, par exemple un DRH et l'épouse de l'employé, et si le DRH change le salaire alors l'épouse aimerait être tenue au courant. Autrement dit l'épouse est une auditrice du changement de la valeur de la propriété liée. Elle doit donc s'enregistrer et être notifiée.

```
import java.awt.event.*;
import java.io.Serializable;
import java.beans.*;

public class Employee implements PropertyChangeListener {
        private int salary;

        // pour déclencher les événements
        private PropertyChangeSupport changesListeners =
                                        new PropertyChangeSupport (this);

        public Employee() {
                this.addPropertyChangeListener(this);
        }

        public void addPropertyChangeListener(PropertyChangeListener p){
                        changesListeners.addPropertyChangeListener (p);
```

```
    }

    public void removePropertyChangeListener(PropertyChangeListener p){
            changesListeners.removePropertyChangeListener (p);
    }

    // Quand le salaire change via setSalary, il faut regarder si la
    // valeur de la propriété a changé et si c'est le cas notifier
    // tous les auditeurs
    public void setSalary (int salary) {
        int oldSalary = this.salary;
        this.salary = salary;

        changesListeners.firePropertyChange ( "salary", oldSalary,
                        this.salary);
    }

    public void propertyChange(PropertyChangeEvent e) {
        if ((e.getSource() == this) &&
                        (e.getPropertyName().equals("salary"))){
            Object o = e.getNewValue();

            int newSalary = (o == null)?
                                this.getSalary()
                                : ((Integer) o).intValue();
            System.out.println("nouveau salaire : " + newSalary);
    }
}
```

Utilisation :

```
public static void main(String [] args) {
   Employee e1 = new Employee();
   e1.setSalary(100000);
   e1.setSalary(200000);
}
```

3.5 Propriétés contraintes

- Elles sont **similaires aux propriétés liée**s.
- En plus d'enregistrer des auditeurs de type **PropertyChangeListener**, le JavaBean enregistre des récepteurs de type **VetoableListener**.
- Avant de changer la valeur d'une propriété, le JavaBean demande aux auditeurs leur autorisation. En cas de refus, l'auditeur génère une exception de type **PropertyVetoException** qui est propagée par la méthode **setPropertyName**.

  Le changement de salaire peut être une propriété contrainte :
  l'épouse ou un auditeur du genre "syndicat des travailleurs" peut refuser une baisse des salaires.

```
public class Employee implements PropertyChangeListener,
                                    VetoableChangeListener {
    ….
    private VetoableChangeSupport vetoesListeners =
                                    new VetoableChangeSupport (this);

    ….
    public void addVetoableChangeListener (VetoableChangeListener v){
        vetoesListeners.addVetoableChangeListener (v);
    }

    public void removeVetoableChangeListener (VetoableChangeListener v){
        vetoesListeners.removeVetoableChangeListener (v);
    }

    public void setSalary (int salary) throws PropertyVetoException{
         int oldSalary = this.salary;
        vetoesListeners.fireVetoableChange ( "salary", oldSalary, salary);

        this.salary = salary;

        changesListeners.firePropertyChange ("salary", oldSalary,
              this.salary);
        }
```

```
    public void vetoableChange(PropertyChangeEvent e)
                                        throws PropertyVetoException{
      ……
      }
}
```

4 Persistance

- La persistance est la capacité d'un objet à mémoriser son état afin d'être recréer ultérieurement.
- Afin d'assurer la persistance d'un objet, Java utilise la **sérialisation**.
- Tout objet désirant être sérialisé doit implémenter le marqueur d'interface défini dans ***java.io.Serializable.java*** :

```
public interface Serializable {
        static final long serialVersionUID = 11966568380767531 33L;
    }
```

- Toutes les variables d'instance doivent être sérialisables. C'est le cas pour les types de base. Pour les autres, leur classe doit implémenter l'**interface Serializable**.
- La sérialisation utilise la méthode

  public final void writeObject(Object obj)

  de la classe ***java.io.ObjectOutputStream***.
- La désérialisation utilise la méthode

  public final Object readObject()

  de la classe ***java.io.ObjectInputStream***.
- La sérialisation sauvegarde toutes les variables et constantes (final) d'instance d'un objet.

Les informations suivantes ne sont pas sérialisées.

Il faut les réinitialiser, si besoin est, lors de la désérialisation :

- les variables d'instance qualifiées **transient** (pertinentes pendant la durée d'exécution seulement).
- les variables et constantes de classe (**static** et **final static**).
- les **images** ne peuvent être sérialisées. Il faut les déclarer transient, les sauver à part et les relire lors de la désérialisation.

Les méthodes suivantes sont à définir pour les données non sérialisées à réinitialiser :

```
private void writeObject(ObjectOutputStream s)
                    throws IOException {
            s.defaultWriteObject();
            // du spécifique si besoin est ...
}

private void readObject(ObjectInputStream s)
    throws ClassNotFoundException, IOException {
            s.defaultReadObject();
            // du spécifique si besoin est …
}
```

Les méthodes ***defaultWriteObject()*** et ***defaultReadObject()*** ne doivent être appellées que dans ce contexte.

Algorithme de sérialisation :

- Si une référence à un autre objet doit être sauvée alors cet objet est sauvé **récursivement,** jusqu'à ce qu'un type de base soit rencontré dans la récursion.
- Si un objet est référencé par plusieurs références alors il est **sérialisé une seule fois**.
- Pour des objets complexes quant à leur structure de données, la question se pose de savoir s'il faut sérialiser toute l'architecture ou ne sauver que les portions nécessaires à la reconstruction lors de la désérialisation.

  Un exemple paradigmatique est celui de la hashtable (revoir un cours classique sur les structures de données et les algorithmes afférents).

  Le cryptage des données est parfois à envisager …

Exemple d'un arbre binaire :

```
TreeNode root = new TreeNode("root");
root.addChild(new TreeNode("left child"));
root.addChild(new TreeNode("right child"));
```

- La sérialisation est assurée par :

```
FileOutputStream fOut = new FileOutputStream("test.out");
ObjectOutput out = new ObjectOutputStream(fOut);
out.writeObject(root);
out.flush();
out.close();
```

- La désérialisation est assurée par :

```
FileInputStream fIn = new FileInputStream("test.out");
ObjectInputStream in = new ObjectInputStream(fIn);
TreeNode root = (TreeNode)in.readObject();
```

Première version :

```
import java.util.*;
import java.io.*;

class TreeNode1 implements Serializable {
        Vector<TreeNode1> childrens;
        TreeNode1 parent;
        String name;
        transient Date date;
```

```
public TreeNode1(String s){
    children = new Vector(5);
    name = s;
    initClass();
  }

 private void initClass () {
    date = new Date();
  }

 public synchronized void setChildren (TreeNode1[] chlidrens) {
    children.removeAllElements();
    int size = childrens.length;
    for (int i = 0; i < size; i++) {
      addChild (chlidrens[i]);
    }
  }

public void setChildren (TreeNode1 element, int position) {
    children.setElementAt (element, position);
    element.parent = this;
  }
```

```
public synchronized TreeNode1[] getChildren () {
        int size = childrens.size();
        TreeNode1 tree[] = new TreeNode1[size];
        for (int i = 0; i < size; i++) {
                tree[i] = (TreeNode1)childrens.elementAt (i);
        }
        return tree;
  }

public TreeNode1 getChildren (int position){
        return (TreeNode1) childrens.elementAt (position);
  }

 public void addChild (TreeNode1 n) {
        children.addElement(n);
        n.parent = this;
  }

@Override
  public String toString() {
            Enumeration e = children.elements();
            StringBuffer buff = new StringBuffer(100);
            buff.append("[ " +  name + " : ");
            while(e.hasMoreElements()){
```

```
                buff.append(e.nextElement().toString());
        }
        buff.append(" ] ");

        return buff.toString();
    }

   public static void main (String args[])
   {
      TreeNode1 root = new TreeNode1("root");
      TreeNode1 left, right, both;
      root.addChild(left = new TreeNode1("left child"));
      root.addChild(right = new TreeNode1("right child"));
      both = new TreeNode1 ("problem child");
      left.addChild (both);
      right.addChild (both);
      System.out.println (root);

try {
 FileOutputStream fOut = new FileOutputStream("test.out");
 ObjectOutput out = new ObjectOutputStream(fOut);
 out.writeObject(root);
 out.flush();
 out.close();
```

```
      FileInputStream fIn = new FileInputStream("test.out");
      ObjectInputStream in = new ObjectInputStream(fIn);
      TreeNode1 n = (TreeNode1)in.readObject();
      in.close();
      System.out.println ("After: " + n);
    }
    catch (Exception e){
      System.out.println ("Whoops");
      e.printStackTrace();
    }
  }
}
```

résultat de l'exécution :

root : [left child : [problem child :]] [right child : [problem child :]]]

After: [root : [left child : [problem child :]] [right child : [problem child :]]]

Seconde version :

- **La date n'est pas sauvegardée lors de la sérialisation mais est réinitialisée à la désérialisation.**
- La fonction d'affichage prend en compte la date.
- Seules les différences avec la version 1 sont données.

```
@Override
public String toString() {
   Enumeration e = children.elements();
   StringBuffer buff = new StringBuffer(100);
   buff.append("[ " +  name + " : ");
   while(e.hasMoreElements()) {
     buff.append(e.nextElement().toString());
   }
   buff.append(" ] ");
   buff.append (date.toString());
   return buff.toString();
  }

 private void writeObject(ObjectOutputStream s)
                                          throws IOException {
        s.defaultWriteObject();
}
```

```
  private void readObject(ObjectInputStream s) throws
                          ClassNotFoundException, IOException {
        s.defaultReadObject();
        initClass();
}
```

Bien que la méthode d'écriture utilise le comportement par défaut, elle est nécessaire dès que la méthode de lecture existe._

résultat de l'exécution :
root : [left child : [problem child :] Wed Sep 15 12:41:04 CEST 1999] Wed Sep 15 12:41:04 CEST 1999[right child : [problem child :] Wed Sep 15 12:41:04 CEST 1999] Wed Sep 15 12:41:04 CEST 1999] Wed Sep 15 12:41:04 CEST 1999
After: [root : [left child : [problem child :] Wed Sep 15 12:41:04 CEST 1999] Wed Sep 15 12:41:04 CEST 1999[right child : [problem child :] Wed Sep 15 12:41:04 CEST 1999] Wed Sep 15 12:41:04 CEST 1999] Wed Sep 15 12:41:04 CEST 1999