Fiche 1 – PropertyChangeSupport

Objectif

Permettre la communication entre Objets sous forme d'événements. On appelle source l'objet émetteur de l'événement.

Packages

```
import java.beans.*;
```

Création d'une classe Listener

Cette classe doit implémenter l'interface `PropertyChangeListener`.
Exemple :

```
public class MastmindPropertyListener implements PropertyChangeListener
```

Implémentation de la méthode de réponse aux événements requise

```
public void propertyChange(PropertyChangeEvent evt) {
        …
    }
```

Création d'un champ `PropertyChangeSupport` dans la source

Exemple :

```
private PropertyChangeSupport changes = new PropertyChangeSupport(this);
```

Encapsulation de la méthode d'enregistrement de listeners dans la source

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

Déclenchement d'un événement dans une propriété de la source

```
public void setXxx(Object obj) {
         xxx = obj;
         changes.firePropertyChange(propertyName,oldValue,newValue);
    }
```

Fiche 2 – Classe Listener Anonyme

Objectif

Définition et enregistrement d'une classe listener anonyme.

Si la classe listener est définie interne à la classe source des événements, ses instances pourront accéder aux autres fonctionnalités de la source

Packages

```
Import java.awt.event.ActionListener;
Import java.awt.event.ActionEvent;
```

Exemple

```
public class MastmindView extends javax.swing.JFrame {
      private javax.swing.JMenuItem jMenuItemTest;
      private javax.swing.JButton jChooseImmageButton;

      private void initComponents() {
            …
            jMenuItemTest = new javax.swing.JMenuItem();
            …
      }

      …
      jMenuItemTest.addActionListener(
            new ActionListener() {
                  public void actionPerformed(ActionEvent evt) {
                        getModel().tester();
                  }
            }
      );

      jChooseImmageButton.addActionListener(
             new ActionListener() {
                  public void actionPerformed(ActionEvent e) {
                              chooseImage();}
             });
      private void chooseImage() { … }
}
```

Fiche 3 – Design pattern Observer

Objectif

Mise en place de la dépendance entre objets. Lorsqu'une source observable est modifiée ses « observers » doivent en être informés pour réagir.

Package

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

Création d'une classe support d'Observable

En général la classe source ne peut hériter de la Classe Observable puisque Java ne possède pas l'héritage multiple. Ajouter un attribut de type Observable dans la classe Source ne convient pas directement puisque une méthode « protected » doit être appelée.

```
public class ObservableSupport extends Observable {
    public ObservableSupport() {
    }
    …
}
```

Dans cette classe support : ajout de la méthode de notification aux « Observers »

```
public void notifyObservers(Object arg) {
        setChanged();
        super.notifyObservers(arg);
    }
```

Cette méthode doit appeler `setChanged()` pour signifier qu'un changement est intervenu et appeler la méthode `notifyObservers` de la superclasse Observable pour réaliser le traitement.

Dans la classe source : déclarer un champ de type `ObservableSupport`

```
private ObservableSupport observable;
```

Dans la classe source : définir l'objet `ObservableSupport`

```
public Observable getObservable() {
        if (observable == null) {
            observable = new ObservableSupport();
        }
        return observable;
    }
```

Dans la classe source : ajouter la méthode d'enregistrement des « observers »

```
public void addObserver(Observer observer) {
        getObservable().addObserver(observer);
}
```

Dans la classe source : notification d'un changement dans un setter d'une propriété

Exemple :

```
public void setProperty(int property) {
        // ici la propriété est de type entier
        this.property = property;
        getObservable().notifyObservers(new Integer(property));
}
```

Test

```
ObservableSource obsc = new ObservableSource();
obsc.addObserver(new SourceObserver());
System.out.println("Property set");
obsc.setProperty(25);
```

Fiche 4 – Design pattern Command (Action)

Objectifs

Des éléments d'interface, comme les item de menu ou les boutons peuvent déclencher les mêmes requêtes en réponse à une action de l'utilisateur.

Package

```
import javax.swing.*;
```

Création d'une classe « action » spécialisant la classe abstraite `AbstractAction`

Cette peut être interne ou externe à la classe source (la classe interface graphique).
Elle doit être associée à la classe cible de l'action (exemple : le modèle de l'interface graphique).
Une hiérarchie plus complète de classes « action » est souvent nécessaire.

```
public class ValidateAction extends AbstractAction {
      private MastmindView source ;
      private Mastmind target;

    /** Creates a new instance of ValidateAction */
    public ValidateAction(MastmindView mmv) {
      source = mmv ;
      target = source.getModel();
    }
    … …
  }
```

Création de la méthode de réponse `actionPerformed` dans la classe « action »

Il est parfois nécessaire d'utiliser des méthodes de la source pour obtenir tous les éléments requis.

```
public class ValidateAction extends AbstractAction {
      … …
  public void actionPerformed(java.awt.event.ActionEvent e) {
       String s2 = source.getActiveColorPropPanel().getColors();
       target.getProp().generate(s2);
       target.evaluate();
  }
}
```

Déclaration et instanciation de l' « action » dans la classe interface graphique

```
public class MastmindView extends javax.swing.Jframe {
      … …
      private ValidateAction validateAction;
      … …
      public ValidateAction getValidateAction() {
        if (validateAction == null)
            validateAction = new ValidateAction(this);
        return validateAction;
      }
}
```

Attachement de l'action sur les éléments d'interface

```
public class MastmindView extends javax.swing.Jframe {
      private javax.swing.JButton jButtonValider;
      private javax.swing.JMenuItem jMenuItemValider;
      … …

      private void initComponents() {
            … …
            jButtonValider.setAction(getValidateAction());
            jMenuItemValider.setAction(getValidateAction());
      }
}
```

Fiche 5 – Design Pattern Singleton

Objectif

Implémentation d'un singleton (mathématiquement : ensemble à un élément)
Un singleton est une classe destinée à n'avoir qu'une seule instance. Cette classe doit gérer cette instance unique.

Déclaration de la classe

Elle doit posséder une variable d'instance de classe privée et une méthode de classe généralement appelée getInstance retournant l'instance unique de la classe.

```
public class MySingleton {
    private static instance;

    /** Creates a new instance of MySingleton */
    public MySingleton () {
        … … …
    }
    public static final MySingleton getInstance() {
        if (instance == null)
            instance = new MySingleton ();
        return instance;
    }
      … … …
}
```

Fiche 6 – Parsing SAX d'un document XML

Objectif

Parsing d'un document XML, et création d'une représentation du document.

Packages

```
import org.xml.sax.*;
import org.xml.sax.helpers.*;
import java.io.*;
```

Sites

Site officiel : http://sax.sourceforge.net/

Exemple de structure d'un document XML bien formé

`<bookmark>`	Ouverture de l'élément racine
`  <moteurs>`	Ouverture de l'élément « moteurs »
`    <url`	Ouverture de l'élément « url »
`      key = "google"`	Attributs key et url
`      url = "http://www.google.com/" >`	> symbole de fin de l'ouverture
`          …`	Contenu de l'élément url
`    </url>`	Fermeture de l'élément url
`    <url`	Nouveau sous-élément de moteurs
`      key = "yahoo"`	
`      url = "http://www.yahoo.com/">…</url>`	
`  </moteurs>`	Fermeture de l'élément « moteurs »
`  <voyage>`	Autre sous-élément de l'élément « bookmark »
`     …`	
`  </voyage>`	
`</bookmark>`	Fermeture de l'élément racine

Principe du parsing SAX (simple API for XML)

- Un objet de type XMLReader lit une source XML (à partir d'un fichier par exemple)
- Il déclenche des événements réceptionnés par un objet de type DefaultHandler (implémentant ContentHandler).
- Les événements correspondent à la lecture de symboles caractéristiques des fichiers XML.
- Il suffit de créer une classe spécialisant DefaultHandler et d'y écrire les méthodes de réponse à ces événements (cela en fonction de la nature de l'application).

Construction de l'XMLReader

Une factory crée l'objet désiré à partir du nom de la classe d'un parser.

```
String xmlReaderClassName = "org.apache.crimson.parser.XMLReaderImpl";
// la librairie Crimson est incluse dans la distribution java
//String xmlReaderClassName = "org.apache.xerces.parsers.SAXParser";
// La librairie Xerces doit être liée au projet
XMLReader xr = XMLReaderFactory.createXMLReader(xmlReaderClassName);
```

Construction du DefaultHandler et rattachement à l'XMLReader

```
// BookmarkHandler est la classe qui hérite de DefaultHandler
BookmarkHandler handler = new BookmarkHandler();
xr.setContentHandler(handler);
xr.setErrorHandler(handler);
```

Parsing d'un fichier

```
// filename est le nom du fichier à parser
FileReader r = new FileReader(filename);
xr.parse(new InputSource(r));
```

Résultat

Le handler lors du parsing crée par exemple un objet qui servira à l'application. Il suffit d'appeler la méthode le renvoyant :

```
… … handler.getBookmarktm();
```

Les lignes précédentes peuvent être regroupées dans une méthode statique d'une classe « Facility » qui renvoie l'objet désiré après le parsing.

Définition de la classe « Handler »

```
public class BookmarkHandler extends DefaultHandler {
    … … …
    /** Creates a new instance of BookmarkHandler */
    public BookmarkHandler() { super(); }
    … … …
}
```

Les principales méthodes à créer dans cette classe sont les suivantes :

```
// début du document
public void startDocument () {
      … … … }

// fin du document
public void endDocument () {
      … … … }

// ouverture d'un élément XML. On récupère son nom et les attributs
public void startElement (String uri, String name,
                  String qName, Attributes atts)  throws SAXException {
      … … … }

// fermeture d'un élément XML. On récupère son nom
public void endElement (String uri, String name, String qName)   {
      … … … }

// caractères rencontrés dans un élément XML
public void characters (char ch[], int start, int length) {
      … … … }
```

Fiche 7 – JTree

Objectifs
Gestion d'un composant graphique de type JTree.

Packages
```
import javax.swing.*;
import javax.swing.tree.*;
import javax.swing.event.*;
```

Composants graphiques
Un composant graphique JTree permet de manipuler graphiquement des arbres.

Modèle
Un JTree est associé à un modèle.
- Java prévoit une interface TreeModel et une classe implémentant cette interface DefaultTreeModel que l'on peut spécialiser.
- Un TreeModel est en fait un nœud racine qui pointe sur les nœuds descendants et ainsi de suite jusqu'aux feuilles de l'arbre.

```
public class BookmarkTreeModel extends DefaultTreeModel {

    /** Creates a new instance of BookMarkTreeModel */
    public BookmarkTreeModel(BookmarkNode node) {
        super(node);
    }
      … … …
}
```

Nœuds
Un arbre est toujours créé avec un nœud (il ne peut pas être vide).
- Java prévoit une interface TreeNode et une classe DefaultMutableTreeNode implémentant cette interface que l'on peut spécialiser pour faciliter la gestion des nœuds d'un arbre.

```
public class BookmarkNode extends DefaultMutableTreeNode {

    /** Creates a new instance of BookmarkNode */
    public BookmarkNode(String name) {
        super(name);
    }
      … … …
}
```

Listener
Sélectionner un élément du composant graphique JTree déclenche l'émission d'un événement à partir duquel il est possible de réagir.
Java défini ainsi l'interface TreeSelectionListener déclarant la méthode valueChange

```
public class BookmarkTreeSelectionListener implements TreeSelectionListener
{
    /** Creates a new instance of TreeListener */
    public BookmarkTreeSelectionListener(… … ) {
        … … …
    }

    public void valueChanged(TreeSelectionEvent e) {
        JTree tree = (JTree) e.getSource();
        … … …
      }
    … … …
}
```

Vue

La vue définit un composant JTree et lui associe des listeners

```
private javax.swing.JTree bookmarkTree;
bookmarkTree = new javax.swing.JTree();

// Association d'un TreeSelectionListener
bookmarkTree.addTreeSelectionListener(
            new BookmarkTreeSelectionListener(this));

// Association d'un MouseListener
bookmarkTree.addMouseListener(new java.awt.event.MouseAdapter() {
      // Pour faire appaître par exemple un popup menu
      public void mousePressed(java.awt.event.MouseEvent evt) {
          showPopup(evt);
      }
      public void mouseReleased(java.awt.event.MouseEvent evt) {
          showPopup(evt);
      }
});
```

Fiche 8 – GridBagLayout

Objectif

Disposition d'éléments dans un panel utilisant le layout : GridBagLayout.

Un GridBagLayout place les composants sur une grille, leur permettant d'occuper plusieurs rangs ou plusieurs colonnes. Les rangs n'ont pas nécessairement la même hauteur et les colonnes n'ont pas nécessairement la même largeur. Un GridBagLayout place les composants dans des cellules rectangulaires utilise les tailles préférées pour déterminer les dimensions des cellules.

Procédé

1. Déclaration du layout :

```
setLayout(new GridBagLayout());
```

2. Déclaration d'un composant :

```
JButton button = new JButton("Valider");
```

3. Déclaration d'un objet contraintes :

```
GridBagConstraints c = new GridBagConstraints();
```

4. Définitions des valeurs des contraintes :

```
c.gridx = 1 ;
c.gridy = 0 ;
…
```

5. Ajout du composant :

```
add(button, c);
```

Et ainsi de suite pour chaque composant. Il est cependant difficile de disposer dans un seul panel un certain nombre de composants. Il est préférable de les répartir dans plusieurs panels. Il est plus simple dans certains cas d'ajouter des éléments invisibles pour disposer correctement les éléments.

Paramètres des contraintes

gridx :	le numéro de la colonne
gridy :	le numéro de la ligne
fill :	le type de remplissage dans la cellule : horizontal, vertical, les deux
gridwith :	le nombre de colonnes occupées
gridheight :	le nombre de lignes occupées
weightx :	le poids d'une colonne (nombre normalement entre 0.0 et 1.0)
weighty :	le poids de la ligne (nombre normalement entre 0.0 et 1.0)

Complément

Voir la page :
http://java.sun.com/docs/books/tutorial/uiswing/layout/gridbag.html