Gestion d'évènements

Principe des évènements

- Java propose des mécanismes de communication entre les objets basés sur des événements.
- Un évènement est émis par un objet (**source**) et reçu par un ou plusieurs objets (**auditeurs**).
  - ➔ On dit que la **source** notifie un événement aux **auditeurs**.
- Pour pouvoir recevoir un (ou des) événement(s), les auditeurs doivent **s'enregistrer** auprès de la (des) source(s).

➔ Principe du patron de conception Observateur

Les évènements

- Les événements sont des héritiers de la classe **`EventObject`**.
  - **`EventObject`** ne propose qu'une méthode **`getSource`** qui renvoie l'objet qui a crée cet événement.
  - Le constructeur de **`EventObject`** admet l'objet qui l'a crée comme seul paramètre.
  - Il est évidement possible de surcharger le constructeur lors de l'héritage en fonction des besoins.
- Par convention, les noms des objets sont de la forme **`XXXEvent`**.
- Le constructeur de la classe **`EventObject`** nécessite la source comme paramètre.

Source et auditeurs

- **La source :** doit être capable de créer des évènements et de les transmettre aux auditeurs.
- **Les auditeurs :** doivent pouvoir s'enregistrer auprès de la source mais peuvent aussi se libérer de cette relation et doivent implémenter une interface qui dérive de **`EventListener`**.
  - En pratique, cette interface a un nom de la forme **`XXXListener`** et doit comprendre une méthode qui admet comme argument un élément de type **`XXXEvent`**.

Mise en œuvre

- Un événement simple peut être construit ainsi :

```
import java.util.EventObject;
public class MonEvenement extends EventObject {
     public MonEvenement(Object source) {
          super(source);
          System.out.println("L'évènement est crée ... ");
     }
}
```

- Les objets voulant recevoir des évènements **`MonEvenement`** doivent implémenter l'interface suivante :

```
import java.util.EventListener;
public interface MonEvenementListener extends EventListener {
     void MonEvenementOccurs(MonEvenement e);
}
```

Mise en œuvre

- Une source d'évènements doit tenir à jour une liste des auditeurs d'événements souvent basée sur un objet de type **`Vector`**.
- Pour la mettre à jour deux méthodes sont crées
  - **`addXXXListener`**
  - **`removeXXXListener`**.
- Dans l'exemple suivant, les événements sont crées à la demande par la méthode **`notifyMonEvenement`**.
- Un évènement (**`evt`**) est crée puis il est transmit à tous les auditeurs (boucle **`for`**...).

Mise en œuvre

```
import java.util.Vector;

public class MaSource {
     private Vector<MonEvenementListener> monEvenementListeners =
                                                  new Vector<MonEvenementListener>();
     public synchronized void addMonEvenementListener(MonEvenementListener mel) {
          if (!monEvenementListeners.contains(mel))
            monEvenementListeners.addElement(mel);
     }
     public synchronized void removeMonEvenementListener(MonEvenementListener mel) {
          if (monEvenementListeners.contains(mel))
            monEvenementListeners.removeElement(mel);
     }
     public void notifyMonEvenement() {
          MonEvenement evt = new MonEvenement(this);
          for (int i = 0; i < monEvenementListeners.size(); i++) {
               MonEvenementListener listener =
                              (MonEvenementListener) monEvenementListeners.elementAt(i);
               listener.MonEvenementOccurs(evt);
          }
     }
}
```

Mise en œuvre

- Pour tester le mécanisme des évènements, on construit des classes qui mettent en œuvre ces différents éléments.
- Une classe auditeur est construite et affichera un simple message quand elle recevra un évènement.

```
public class MaClasseAuditeur implements MonEvenementListener {
     public void MonEvenementOccurs(MonEvenement e) {
        System.out.println("J'ai créé un evenement de " + e.getSource());
     }
}
```

Mise en œuvre

- La classe exécutable contient deux objets : une source (classe **`MaSource`**) et un auditeur (classe **`MaClasseAuditeur`**) :

```
public class TestEvenement {
     public static void main(String[] args) {
          MaSource uneSource = new MaSource();
          MaClasseAuditeur unAuditeur = new MaClasseAuditeur();
          uneSource.addMonEvenementListener(unAuditeur);
          uneSource.notifyMonEvenement();
     }
}
```

- A l'exécution on obtient le résultat suivant :

```
L'évènement est crée ...
J'ai crée un evenement de MaSource@1cfb549
```

Gestion des événements dans Swing

- Tous les composants de Swing (et de AWT) créent des évènements en fonction des actions de l'utilisateur.
- Les évènements sont des descendants de **`EventObject`**, noté **`XXXEvent`**.
- Les composants proposent des méthodes du type **`AddXXXListener`** pour enregistrer un auditeur.
- L'auditeur doit implémenter l'interface correspondante qui est de la forme **`XXXListener`**.

Gestion des événements dans Swing

- **Exemple :** le composant `JAbstractButton` (et donc tous ses descendants) possède une méthode `addActionListener` utilisée pour enregistrer une classe implémentant l'interface `ActionListener`.
- L'interface `ActionListener` n'a qu'une méthode `actionPerformed(ActionEvent e)` qui doit être implémentée pour gérer l'évènement (l'évènement est disponible via le paramètre `e`).
- La classe `ActionEvent` propose plusieurs méthodes utiles comme `getSource` pour identifier la source de l'évènement.

Les différents évènements

- Swing dispose de deux types d'évènements :
  - **Evènements de base :** communs à tous les descendants de Component (et donc de `JComponent`), couvrent les évènements bas-niveau comme les mouvements de la souris, la modification des composants ou l'utilisation du clavier.
  - **Evènements sémantiques :** représentent des actions de haut-niveau de l'utilisateur comme la sélection d'un menu, la modification du curseur dans les composants textuels, ...

Evènements de base

Evènements	Description
`ComponentEvent`	Un composant a bougé, changé de taille, changé de visibilité.
`ContainerEvent`	Un composant a été ajouté/enlevé du conteneur.
`FocusEvent`	Le composant a gagné/perdu le focus.
`KeyEvent`	Une touche a été appuyée/relâchée.
`MouseEvent`	La souris a bougée ou l'un des boutons a changé d'état.
`MouseWheelEvent`	La molette de souris a été tournée.
`WindowEvent`	La fenêtre a été réduite/redimensionnée/agrandie.

Evènements sémantiques

Evènements	Description
`ActionEvent`	De nombreux éléments génèrent cet événement : un **`JButton`** lors d'un clic de souris, un élément de menu (**`JXXXMenuItem`**) est sélectionné, la touche Entrée a pressée dans **`JTextField`**,. . .
`ChangeEvent`	Un changement d'état a eu lieu dans un composant : un élément radio (bouton ou menu), une case à cocher (bouton ou menu), un bouton ou un élément de menu a changé d'état, une glissière a bougée, changement d'onglet dans un **`JTabbedPane`**,. . .
`CaretEvent`	Modification de la position du curseur dans un élément **`JTextComponent`** ou un de ses descendants.
`ItemEvent`	Un nouvel élément a été sélectionné dans une **`JComboBox`**.
`ListSelectionEvent`	Un nouvel élément a été sélectionné dans une **`JList`**.
`...`	**...**

Gestion des évènements par la classe graphique

- L'approche la plus simple de mise en œuvre.

```
import java.awt.event.*;
import javax.swing.*;
public class TestEvent1 extends JPanel implements ActionListener, KeyListener {
     JButton leBouton;
     JTextField leTextField;
     public TestEvent1() {
          leBouton.addActionListener(this);
          leTextField.addKeyListener(this);
     }
     public void actionPerformed(ActionEvent e) {   }

     public void keyTyped(KeyEvent e) {   }

     public void keyPressed(KeyEvent e) {    }

     public void keyReleased(KeyEvent e) {    }
}
```

- Cette méthode est très simple lorsque le nombre d'évènements à gérer est faible.

Gestion des évènements par la classe graphique

- Si deux (ou plus) composants émettent le même évènement, il faut identifier la source de l'évènement dans la procédure concernée à l'aide de **`getSource`**.

```
import java.awt.event.*;
import javax.swing.*;
public class TestDeuxBouton extends JPanel implements ActionListener {
     JButton leBouton1, leBouton2;
     public TestDeuxBouton() {
          leBouton1.addActionListener(this);
          leBouton2.addActionListener(this);
     }
     public void actionPerformed(ActionEvent e) {
          if (e.getSource() == leBouton1) {
            // La source est leBouton1
          }
          if (e.getSource() == leBouton2) {
            // La source est leBouton2
          }
     }
}
```

Gestion des évènements par des classes dédiées

- Consiste à construire une classe par auditeur d'évènement.

```
public class ListenerBouton1 implements ActionListener{
     public void actionPerformed(ActionEvent e) {
        // Partie utile
     }
}
```

```
public class ListenerBouton2 implements ActionListener{
     public void actionPerformed(ActionEvent e) {
        // Partie utile
     }
}
```

```
public class TestClassesExternes extends JPanel {
     JButton leBouton1, leBouton2;
     public TestClassesExternes() {
          leBouton1 = new JButton(" Bouton 1");
          leBouton2 = new JButton(" Bouton 2");
          leBouton1.addActionListener(new ListenerBouton1());
          leBouton2.addActionListener(new ListenerBouton2());
     }
}
```

Gestion des évènements par des classes dédiées

- Les deux auditeurs sont instanciés dans les méthodes d'enregistrement car aucune référence n'est nécessaire.
- Les classes auditeurs n'ont pas directement accès aux variables membres de la classe visuelle.
- Ils est donc nécessaire de créer des références entre ces deux éléments.

Gestion des évènements par des classes membres internes

- Le problème évoqué avant peut être résolu en utilisant des classes membres internes.
- Les classes membres peuvent accéder facilement aux éléments de la classe graphique.
- Cette méthode conduit à créer une classe membre interne par composant et par évènement.

Gestion des évènements par des classes membres internes

```
import java.awt.event.*;
import javax.swing.*;
public class TestClassesMembres extends JPanel {
     JButton leBouton1, leBouton2;
     public TestClassesMembres() {
          leBouton1 = new JButton("Bouton 1");
          leBouton2 = new JButton("Bouton 2");
          leBouton1.addActionListener(new ListenerBouton1());
          leBouton2.addActionListener(new ListenerBouton2());
     }
     private class ListenerBouton1 implements ActionListener {
          public void actionPerformed(ActionEvent e) {
            // Partie utile
          }
     }
     private class ListenerBouton2 implements ActionListener {
          public void actionPerformed(ActionEvent e) {
            // Partie utile
          }
     }
}
```

Gestion des évènements par des classes membres internes anonymes

- Dans le cas des auditeurs, les classes sont simplement instanciées pour être ensuite utilisée comme paramètre de la fonction **`addXXXListener`**.
- Les instanciations inutiles sont supprimées et chaque auditeur est rattaché à un seul objet.
- Cette approche permet de facilement modifier le code, si un on ajoute (ou on enlève) un composant ou/et un évènement.

Gestion des évènements par des classes membres internes anonymes

```
public class TestClassesAnonymes extends JPanel {
     JButton leBouton;
     JTextField leTextField;
     public TestClassesAnonymes() {
     leBouton = new JButton(" Bouton 1");
     leTextField = new JTextField();
     leBouton.addActionListener(new ActionListener() {
          public void actionPerformed(ActionEvent e) {
            // Partie utile
          }
          });
     leTextField.addKeyListener(new KeyListener() {
          public void keyTyped(KeyEvent e) {
            // Partie utile
          }
          public void keyPressed(KeyEvent e) {
            // Partie utile
          }
          public void keyReleased(KeyEvent e) {
            // Partie utile
          }
          });
     }
}
```

Gestion des évènements par des classes d'adaptations

- Les interfaces auditeurs les plus utilisées comportent de nombreuses méthodes.
  - ➔ Lors de la conception d'un auditeur on doit donc déclarer toutes ces méthodes mêmes si elles ne sont pas utilisées.
  - ➔ Ce développement prend du temps et alourdit le code.
  - ➔ Afin d'éviter ceci, on peut utiliser les classes d'adaptation (ou adaptateurs factices).
- Les classes d'adaptation sont des classes abstraites qui contiennent toutes les méthodes de l'interface auditeur d'évènement.

Gestion des évènements par des classes d'adaptations

- **Exemple :** pour l'interface **`KeyListener`** on doit définir les trois méthodes **`keyTyped`**, **`keyPressed`** et **`keyReleased`**.
- Si seul l'évènement **`keyTyped`** est utilisé, le code suivant est nécessaire.

```
import java.awt.event.*;
public class ListenerClavier implements KeyListener {
     public void keyTyped(KeyEvent e) {
        // Partie utile
     }
     public void keyPressed(KeyEvent e) {
     }
     public void keyReleased(KeyEvent e) {
     }
}
```

Gestion des évènements par des classes d'adaptations

- La classe abstraite **`KeyAdapter`** implémente les trois méthodes de la manière suivante :
  - **`public void keyTyped ( KeyEvent e) {}`**
  - **`public void keyPressed( KeyEvent e) {}`**
  - **`public void keyReleased( KeyEvent e) {}`**
- Pour construire l'auditeur on crée une classe fille de la classe **`KeyAdapter`** qui surcharge les méthodes utiles.
- Ce qui conduit à l'implémentation suivante :

```
import java.awt.event.*;

public class ListenerClavierBis extends KeyAdapter {
     public void keyTyped(KeyEvent e) {
        // Partie utile
     }
}
```

Gestion des évènements par des classes d'adaptations

- La dernière méthode de conception a toutefois une limite :
  - ➔ Java ne supportant pas l'héritage multiple, on doit donc multiplier le nombre de classes auditeurs.
- Par exemple, un auditeur qui implémente les interfaces **`KeyListener`** et **`MouseListener`** peut avoir la déclaration suivante :

```
import java.awt.event.*;
public class ListenerMultiple implements KeyListener, MouseListener {
     public void keyTyped(KeyEvent e) {
        // Partie utile
     }
     public void keyPressed(KeyEvent e) { }
     public void keyReleased(KeyEvent e) { }
     public void mouseClicked(MouseEvent e) {
        // Partie utile
     }
     public void mousePressed(MouseEvent e) { }
     public void mouseReleased(MouseEvent e) { }
     public void mouseEntered(MouseEvent e) { }
     public void mouseExited(MouseEvent e) { }
}
```

Gestion des évènements par des classes d'adaptations

- Cet auditeur peut être utilisé de la manière suivante :

```
ListenerMultiple unListener = new ListenerMultiple();
...
unComposant. addMouseListener( unListener);
unComposant. addKeyListener( unListener);
...
```

- Pour utiliser les adaptateurs, on doit créer deux classes **`listeners`**, l'une pour **`KeyListener`** et l'autre pour **`MouseListener`**.

```
public class ListenerClavierBis extends KeyAdapter {
     public void keyTyped(KeyEvent e) {
        // Partie utile
     }
}
```

```
public class ListenerSouris extends MouseAdapter {
     public void mouseClicked(MouseEvent e) {
        // Partie utile
     }
}
```

Gestion des évènements par des classes d'adaptations

- Ces **`listeners`** pourront être utilisés ainsi :

```
...
ListenerClavier unListenerClavier = new ListenerClavier();
ListenerSouris unListenerSouris = new ListenerSouris();
...
unComposant. addMouseListener( unListenerSouris);
unComposant. addKeyListener( unListenerClavier);
```

- Les classes adaptateurs peuvent être utilisées pour créer des classes auditeurs externes comme ci-dessus, mais aussi des classes membres internes ou des classes membres anonymes.

```
unComposants.addMouseListener(new MouseAdapter(){
                                    public void mouseClicked( MouseEvent e) {
                                        // Partie utile
                                    }
                                });
```

- Cette approche est la plus utilisée, notamment par les environnements de développement.

Evènements de fenêtre

- Il existe deux `listeners` pour récupérer les évènements de fenêtre (icônification, fermeture, etc)
- `WindowStateListener`
  - `windowStateChanged()`, changement d'état
- WindowListener
  - `windowOpened()`, ouverture
  - `windowClosing()`, clique sur la croix
  - `windowClosed()`, fermeture
  - `windowActivated()`, activé (avant-plan)
  - `windowDeactivated()`, désactivé
  - `windowIconified()`, icônification
  - `windowDeiconified()`, dé-icônification

Evènements de fenêtre (2)

- La classe **WindowAdapter** implante l'interface **WindowListener**, chaque méthode ne fait rien.

```
public class WindowExample {
  static class Closer extends WindowAdapter {
    public WindowCloser(JFrame frame) {
      this.frame=frame;
    }
    public void windowClosed(WindowEvent e) {
      System.out.println("arg, je meurs");
    }
    public void windowClosing(WindowEvent e) {
      frame.dispose();
    }
    private final JFrame frame;
  }
  public static void main(String[] args) {
    final JFrame frame=new JFrame("WindowExample");
    frame.addWindowListener(new Closer(frame));
    frame.setSize(400,300);
    frame.setVisible(true);
  }
}
```

Libère la ressource de la plateforme