Gestion événementielle

plan

- La programmation événementielle
  - Le design pattern observer
  - La mise en œuvre avec Java
- Evenements graphiques
  - Introduction
  - Event Dispatch Thread
  - Partage de ressources
  - SwingWorker
  - Événements souris
  - Événements clavier

La programmation événementielle

Le design pattern observer

La majorité des problèmes classiques en programmation ont donné lieu à un design pattern ou « modèle de conception ». Autrement dit, il existe une méthode “standard” pour traiter chaque problématique classique de programmation.

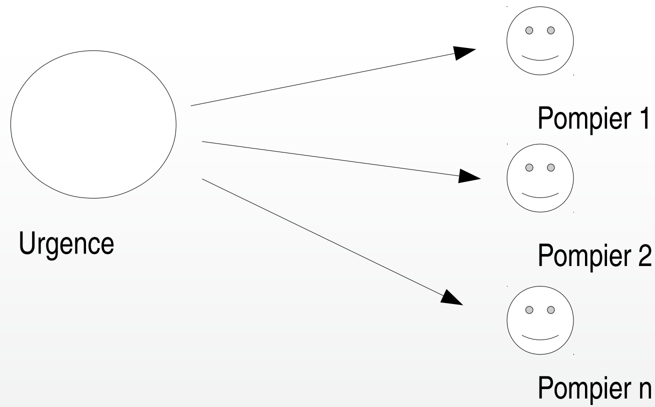
Le design pattern “Observer” permet de répondre au besoin de “dialogue” qui peut exister entre différents objets.

La programmation événementielle

Prenons l'exemple d'un Objet Pompier et d'un Objet Urgence.

Les pompiers doivent être informés lorsqu'une urgence survient.

Il y a une relation “d'observation“ entre les deux.



La programmation événementielle

L'objet dit « observable » est l'urgence, l'objet dit « observateur » est le pompier.

Dans le design pattern « Observer », l'objet observable va « contacter » tous les observateurs « abonnés » lorsque cela est nécessaire.

C'est donc l'observable qui pousse l'information ce qui implique que ce dernier doit connaître ses observateurs.

La programmation événementielle

Si l'on traduit les entités Urgence et Pompier en Java selon ce design pattern, on obtient par exemple :

```
public class Pompier {
      private String nomPompier;

      public Pompier(String nom){
            this.nomPompier=nom;
      }
      public void intervientSurUrgence(){
            System.out.println("Je m'appelle "+nomPompier+" et j'intervient sur l'urgence");
      }
}
```

```
public class Urgence {

      private List<Pompier> lesPompiersDeService=new ArrayList<Pompier>();

      public void ajoutePompier(Pompier p){
            lesPompiersDeService.add(p);
      }

      public void caUrge(){
            for (Pompier p : lesPompiersDeService){
                  p.intervientSurUrgence();
            }
      }
}
```

Exercice n°1

- Voir document dédié

La programmation événementielle

Les pompiers interviennent sur un large panel d'urgence, il serait intéressant de pouvoir informer le pompier de la nature de l'urgence :

- Incendie
- Accident etc....

Le design pattern observer parle alors d'événement (Event).

La programmation événementielle

On peut adapter le Pompier et l'urgence en tenant compte de cet événement :

```
public class Urgence {

...

     public void incendie(){
          for (Pompier p : lesPompiersDeService){
                    p.intervientSurUrgence("INCENDIE");
          }
     }
     public void accident(){
          for (Pompier p : lesPompiersDeService){
                    p.intervientSurUrgence("ACCIDENT");
          }
     }
}
```

```
public class Pompier {

     ...

     public void intervientSurUrgence(String event){
          System.out.println("Je m'appelle "+nomPompier+" et j'intervient sur l'urgence "+event);
     }
}
```

Exercice n°2

- Voir document dédié

La mise en œuvre avec Java

Java propose une manière standard de gérer ces 3 concepts :

- Observateur
- Observable
- Événement

La programmation événementielle

L'élément Observateur sera qualifié de `Listener` (écouteur) et implémentera systématiquement l'interface `EventListener`.

Cette interface possède déjà dans l'API Java une multitude d'implémentations :

`http://java.sun.com/j2se/1.4.2/docs/api/java/util/EventListener.html`

La programmation événementielle

D'autre part, l'événement ne sera pas un simple `String` mais un objet qui héritera de la classe `EventObject`. Il devra obligatoirement déclarer un constructeur prenant en paramètre l'objet source de l'événement (la source est l'objet à l'origine de l'événement).

http://java.sun.com/j2se/1.4.2/docs/api/java/util/EventObject.html

```
public class MyEvent extends EventObject{

    public MyEvent(Object source){
        super(source);
    }

}
```

Dans le constructeur, on va faire appel au constructeur du parent

La programmation événementielle

L'élément observable quand à lui peut être de toute nature. Cependant, traditionnellement on lui ajoutera au minimum la possibilité :

- D'abonner un `Listener` (équivalent au `ajoutPompier`)

- De désabonner un `Listener`

Exercice n°3

- Voir document dédié

Événements graphiques

Introduction

Parmi tout les Listeners déjà inclus dans l'API Java, il y a bien-sûr ceux liés aux actions sur les composants graphiques.

Nous avons par exemple la possibilité d'abonner un `Listener` à l'événement de « Click » sur un `JButton`. On utilisera alors un `Listener` de type `ActionListener` :

```
public class MyActionListener implements ActionListener {

    public void actionPerformed(ActionEvent evt){
        System.out.println("Bouton clické !");
    }
}
```

```
MyActionListener mal=new MyActionListener();

bouton2.addActionListener(mal);
```

Exercice n°4

- Voir document dédié

Événements graphiques

Imaginons que l'opération effectuée par le `Listener` soit une opération longue (disons lire un gros fichier). Que va t-il se passer ?

Et bien durant toute la durée de l'opération, votre fenêtre et son contenu seront complètement bloqués !

Exercice n°5

- Voir document dédié

Événements graphiques

Event Dispatch Thread

Pendant 30 secondes l'action que vous avez effectuée sur le bouton 1 ne sera pas prise en compte.

Pourquoi ? Et bien parce que tout ce qui concerne l'affichage graphique et la gestion d'événement se passe dans le même thread : l' « Event Dispatch Thread ».

L« Event Dispatch Thread » est un Thread particulier propre à Swing. Vous n'avez pas à créer ou gérer ce Thread, il existe à côté de votre Thread principal quoi qu'il arrive.

Ce dernier a la charge de séquencer les événements qui se passent au niveau des composants, que se soit des événements naturels de rafraîchissement déclenchés par Swing ou que ce soit des événements utilisateur (cliquer sur un bouton).

Événements graphiques

Pour y remédier, nous aurions tous le réflexe naturel de créer un autre Thread pour exécuter notre tâche longue.

```
Runnable r=new Runnable() {
     public void run() {
          try {
               Thread.sleep(30000);
          } catch (InterruptedException e) {
               e.printStackTrace();
          }
     }
};

Thread t=new Thread(r);
t.start();
```

Exercice n°6

- Voir document dédié

Événements graphiques

Partage de ressources

Cette solution n'est cependant pas parfaite, nous allons voir pourquoi.

Exercice n°7

- Voir document dédié

Événements graphiques

Nous avons donc 2 boutons qui ont pour objet de modifier un libellé.

Or ce que nous faisons sur le libellé suite au click sur le bouton 2 se passe dans un Thread particulier et non pas dans l'Event Dispatch Thread.

Dans notre exercice le libellé devient une ressource partagée et nous pouvons rencontrer des problèmes de concurrence d'accès.

La modification initiée par le bouton 2 devrait donc être effectuée dans le processus normal de l'Event Dispatch Thread pour assurer la séquentialité.

Nous y parviendrons grâce à la méthode `invokeLater` de `SwingUtilities`.

```
Runnable r=new Runnable() {
     public void run() {
          try {
               Thread.sleep(20000);
          } catch (InterruptedException e) {
               e.printStackTrace();
          }
          SwingUtilities.invokeLater(new Runnable(){
               public void run(){
                    label.setText("Tache longue terminée");
               }
          });
     }
};

Thread t=new Thread(r);
t.start();
```

Exercice n°8

- Voir document dédié

Événements graphiques

Le code devient plutôt complexe à réaliser et pourtant il s'agit de situations très courantes !

Les actions utilisateurs se terminent généralement par une mise à jour graphique !

Événements graphiques

SwingWorker

Heureusement l'API Java propose une classe d'aide pour nous simplifier l'écriture, la class `SwingWorker`.

Cette classe encapsule les 2 opérations que l'on cherche à effectuer à la suite d'un événement utilisateur.

Pour cela on dispose de 2 méthodes :

- La méthode `protected Object doInBackground()` à l'intérieur de laquelle on pourra ajouter notre traitement métier parfois long.

- La méthode `protected void done()` à l'intérieur de laquelle on effectuera les mises à jour graphique. Chose importante, les mises à jour s'effectueront implicitement dans l'Event Dispatch Thread sans que l'on ait à utiliser `invokeLater`.

Pour démarrer l'instance de `SwingWorker`, on utilise la méthode `execute()`

Événements graphiques

Le code devient le suivant :

```
...
    public void actionPerformed(ActionEvent evt) {

        MyWorker worker=new MyWorker(label);

        worker.execute();

    }
...
```

```
public class MyWorker extends SwingWorker{

    private JLabel label;

    public MyWorker(JLabel label){
        this.label=label;
    }

    protected Object doInBackground() throws Exception {
        try {
            Thread.sleep(20000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        return null;
    }

    protected void done() {
        super.done();
        label.setText("Tache longue terminée");
    }

}
```

Exercice n°9

- Voir document dédié

Événements graphiques

On remarque que la méthode `doInBackground` peut retourner un résultat.

Il s'agit de l'éventuel résultat obtenu suite à votre traitement métier.

Imaginez par exemple que l'on calcule le nombre premier le plus proche de 100000, on souhaitera bien-sur récupérer ce résultat quelque part.

Pour récupérer le résultat calculé par `doInBackground`, on fera appel à la méthode `get` de notre `SwingWorker`. Celle-ci attendra que `doInBackground` se termine avant de retourner quoi que ce soit.

Événements graphiques

On pourrait donc théoriquement mettre en oeuvre le code suivant.

```
...
    protected Object doInBackground() throws Exception {
        return nextNumberAfter1000();
    }

...
```

```
...
    public void actionPerformed(ActionEvent evt) {

        MyWorker worker=new MyWorker(label);

        worker.execute();

        label.setText("Le resultat est "+worker.get())

    }
...
```

Cependant nous bloquerions alors à nouveau l'EDT car la méthode `get()` ne donnera un résultat qu'à la fin du traitement de `doInBackground` !

Événements graphiques

Heureusement la classe `SwingWorker` dispose d'une propriété `state` qui se met à jour lorsque `doInBackground` se termine (notamment) et nous pouvons abonner un Listener au changement de valeur de cette propriété.

Nous récupérerons alors le résultat dès lors que le `SwingWorker` à terminé sa méthode `doInBackground` (méthode `isDone()`) :

```
...
     public void actionPerformed(ActionEvent evt) {

          MyWorker worker=new MyWorker(label);
          worker.addPropertyChangeListener(this);

          worker.execute();

          //label.setText("Le resultat est "+worker.get())
     }
     public void propertyChange(PropertyChangeEvent arg0) {

          MyWorker wk=(MyWorker)arg0.getSource();

          if (wk.isDone()) {
               Object result = null;
               try {
                    result = (Integer) worker.get();
               } catch (Exception e) {
                    e.printStackTrace();
               }
          }
     }
...
```

Événements graphiques

La tâche longue étant désormais une tâche de fond, il est intéressant de pouvoir montrer à l'utilisateur que celle-ci progresse. `SwingWorker` propose également une propriété `progress` auquel il est possible de s'abonner.

Il sera bien évidemment nécessaire de faire évoluer la valeur de cette propriété qui est exprimée en pourcentage.

```
...
     protected Object doInBackground() throws Exception {

          for (int i=0;i<5;i++){
               try {
                    Thread.sleep(1000);
               } catch (InterruptedException e) {
                    e.printStackTrace();
               }
               setProgress(100 * (i+1) / 5);
          }
          return new Integer(1);//Résultat quelconque pour l'exemple
     }
...
```

```
...
     public void propertyChange(PropertyChangeEvent arg0) {

          if (arg0.getPropertyName().equals("progress")){
               System.out.println("avancement "+arg0.getNewValue()+"%");
          }

     }
...
```


Traditionnellement on plutôt alors à jour une barre de progression.

Exercice n°10

- Voir document dédié

Événements graphiques

Note : On peut encore aller plus loin. Il est également possible de traiter des résultats temporaires sans attendre la fin de `doInBackground()`.

Les résultats temporaire seront fournit dans `doInBackground` via la méthode `publish` de `SwingWorker`.

Ces résultats seront reçus par la méthode `process` de `SwingWorker`.

Événements graphiques

Événements souris

Parmi les listeners disponibles dans l'API Java, il y a ceux qui concernent la souris.

Il existe plusieurs interfaces. Certaines sont spécialisées dans des domaines précis, d'autres sont plus généralistes.

`MouseListener` permet de gérer les actions standards de la souris telles que les différents clics (bouton enfoncé, bouton relâché) ou le survol d'un élément.

`MouseMotionListener` gère les évènements liés au mouvement de la souris, drag y compris (Le drag cible les mouvements click maintenu).

`MouseWheelListener` gère les évènements liés à la roulette de la souris.

Événements graphiques

Une fois le choix de l'interface à implémenter effectué, il suffit d'ajouter à chaque composant le listener approprié.

```
myJLabel.addMouseWheelListener(new MyMouseWheelListener());
```

Les méthodes à notre disposition sont :

`addMouseWheelListener` : pour abonner un listener de type `MouseWheelListener` au composant

`addMouseMotionListener` : pour abonner un listener de type `MouseMotionListener` au composant

`addMouseListener` : pour abonner au listener de type `MouseListener` au composant

Événements graphiques

`MouseListener` est une interface qui permet de gérer les actions standards de la souris telles que les différents clics (bouton enfoncé, bouton relâché), le survol d'un élément.

Les méthodes à implémenter sont :

`mousePressed` : déclenché lorsque l'on enfonce un bouton de la souris

```
public void mousePressed(MouseEvent e) {}
```

`mouseReleased` : déclenché lorsque l'on relâche le bouton

```
public void mouseReleased(MouseEvent e) {}
```

Événements graphiques

`mouseEntered` : déclenché lorsque l'on entre dans la surface occupée par le composant

```
public void mouseEntered(MouseEvent e) {}
```

`mouseExited` : déclenché lorsque l'on quitte une surface occupée par le composant

```
public void mouseExited(MouseEvent e) {}
```

`mouseClicked` : déclenché lorsque l'on clique sur un bouton, quel que soit le bouton clické

```
public void mouseClicked(MouseEvent e) {}
```

Événements graphiques

`MouseMotionListener` est une interface qui permet de gérer les évènements liés au mouvement de la souris et du drag.

Les méthodes à implémenter sont :

`mouseDragged` : déclenché lorsque l'on « attrape » un élément

```
public void mouseDragged(MouseEvent me) {}
```

`mouseMoved` : déclenché lorsque l'on bouge la souris

```
public void mouseReleased(MouseEvent e) {}
```

Événements graphiques

`MouseWheelListener` est une interface qui permet de gérer les évènements liés à la roulette de la souris.

Les méthodes à implémenter sont :

`mouseWheelMoved` : déclenché lorsque l'on «actionne» la roulette

```
public void mouseWheelMoved(MouseWheelEvent mwe) {}
```

Événements graphiques

Lorsque l'on gère les événements souris, on ne souhaite généralement par traiter tous les événements, mais plutôt une sélection d'entre-eux. Or lorsque l'on implémente `MouseListener`, `MouseMotionListener` ou `MouseWheelListener` on se retrouve dans l'obligation de compléter toutes les méthodes de l'interface.

En outre, il arrive aussi très fréquemment que les événements que l'on souhaite gérer proviennent de plusieurs listeners, ce qui multiplie encore le nombre d'interfaces à compléter.

Événements graphiques

Pour nous simplifier la vie, on fait généralement appel à des adapteurs. Il s'agit de classes qui implémentent « à vide » toutes les méthodes d'une interface, laissant le développeur réécrire celles qui l'intéressent réellement.

Dans le cas des événements souris, il existe 2 types d'adapteurs.

`MouseInputAdapter` implémente toutes les méthodes de `MouseListener` et de `MouseMotionListener`

`MouseAdapter` implémente toutes les méthodes de `MouseListener`, de `MouseMotionListener` et de `MouseWheelListener`.

Exercice n°11

- Voir document dédié

Événements clavier

L'API java propose des listeners pour interagir avec le clavier. Il s'agira alors d'implémenter l'interface `KeyListener`. Les méthodes disponibles sont les suivantes :

```
public void keyTyped(KeyEvent ke) {}

public void keyPressed(KeyEvent ke) {}

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

`keyPressed` : lorsque la touche est enfoncée

`KeyReleased` : lorsque la touche est relâchée

`keyTyped` : lorsqu'un caractère est produit (il peut s'agir de la combinaison de plusieurs touches ex: « ê »)

Les composants s'abonneront à ce type de Listener par le biais de la méthode `addKeyListener`.