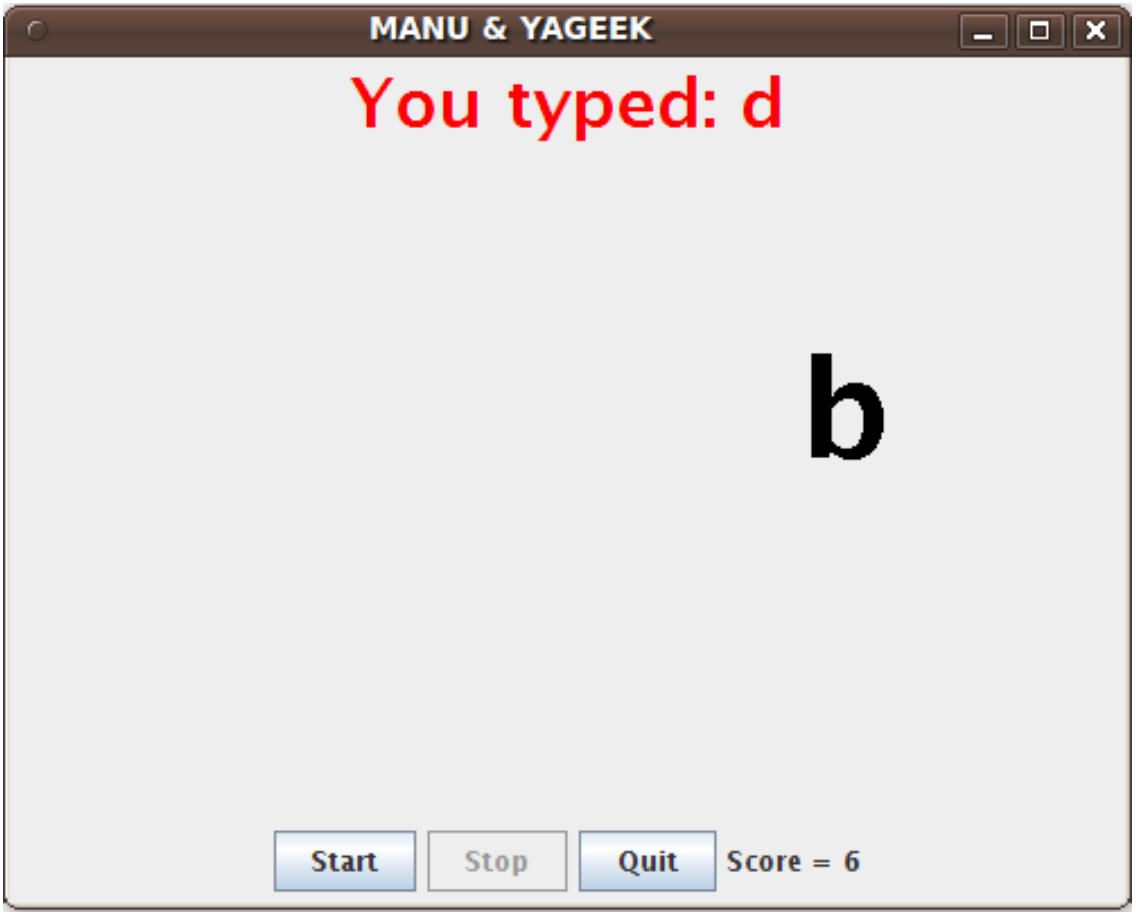
Projet Jeu de caractères



Le 10 janvier 2010

Par Yannick Heinrich et Emmanuel Roussel - GE5S

Table des matières

I Les différentes étapes

A Afficher un caractère entré au clavier

Dans cette première étape, nous affichons un caractère entré à partir du clavier dans une fenetre graphique.

FIG. 1 – La première étape : afficher le caractère entré

Nous avons créé 4 classes :
- Main : contient la fonction main, et crée la fenetre MainFrame.
- MainFrame : Hérite de JFrame. Affiche le JPanel PanelCarac. Contient un écouteur sur keyPressed.
- EvtCarac : Classe observable. Utilise les pattern Observer et Singleton.
- PanelCarac : JPanel qui permet l'affichage du caractère. Il observe EvtCarac, et se met donc à jour dès que EvtCarac lui fait signe.

Lorsqu'on lance le programme, trois threads sont démarrés :
- Le thread principal, qui exécute la fonction main
- Le thread de gestion de la mémoire (ramasse-miette, ou garbage collector)
- L'EventDispatchThread, qui s'occupe de l'interface graphique.

Tout ce qui concerne l'interface graphique doit etre exécuté dans ce thread. Pour cela on dispose d'une queue d'évenements. Les actions sont effectuées de manière séquentielle par ce thread (cela permet d'éliminer les problèmes de synchronisation).
Ainsi, dans la structure de notre programme, nous avons une classe Main qui contient une fonction main qui est le point d'entrée de notre programme.
Cette fonction main va mettre la création de notre fenetre dans la liste des actions à effectuer par l'EventDispatchThread, grace à l'appel à la fonction invokeLater.

```
java.awt.EventQueue.invokeLater(new Runnable()
public void run()
new MainFrame().setVisible(true);

);
```

L'EventDispatchThread s'occupe de l'affichage des fenetres et de leur rafraichissement. Les écouteurs d'évemenements sur la fenetre sont aussi exécutés dans ce thread.
Les evenements de type KeyPressed sont donc pris en charge par ce thread.

B Générateur aléatoire de caractères

Dans cette deuxième étape, on génère aléatoirement un caractère et on l'affiche dans la fenetre graphique.

Le caractère se déplace à une vitesse aléatoire de gauche à droite.

FIG. 2 – La deuxième étape : génération aléatoire et déplacement.

Pour réaliser cela, on utilise la classe Timer de Swing (par la suite, on créera notre propre timer) :

```
ActionListener taskPerformer = new ActionListener()
public void actionPerformed(ActionEvent evt)
positionCaract+=5 ;
repaint() ;

;
tim = new Timer(time, taskPerformer) ; //timer pour exécuter la fonction à interval de temps régulier.
tim.start() ;
```

On incrémente donc la position du caractère à interval de temps régulier. Cet interval est choisi aléatoirement à chaque génération d'un nouveau caractère :

```
time=(int)(Math.random()*30+10) ; //vitesse aléatoire
```

Il y a des données partagées. La variable positionCaract, qui est la position du caractère dans la fenetre, est accédée par le thread du timer et par l'EventDispatchThread pour mettre à jour l'affichage. En effet, la méthode paintComponent du JPanel contient le code :

```
if(positionCaract >= this.getWidth())
positionCaract=0 ;
evtCarac.generateCarac() ;
```

Cependant ce n'est pas génant, puisque l'accès en lecture de la variable pour la condition est fait de manière atomique. De meme, l'affectation de 0 à la variable est également atomique.
La variable n'étant qu'incrémentée par le timer, meme si la valeur de positionCaract est changée entre l'entrée dans la condition et l'affectation, la condition reste vraie.
On ne peut donc pas ici avoir de problèmes entre les threads.

C Boutons start, stop et quit

Dans cette troisième étape, on rajoute les trois boutons.
Par ailleurs, nous avons développé notre propre timer plutot que d'utiliser la classe Swing.Timer toute faite. Il s'agit de la classe TimerPerso.
La classe est abstraite, et contient la définition de la fonction abstraite *public abstract void iteration() ;*

Pour créer un nouveau timer, on procède donc comme suit :

```
tim = new TimerPerso(time, true)
@Override
```

```
public void iteration()
positionCaract+=5 ;
repaint() ;

;
tim.start() ; //on démarre le timer.
```

Nous avons implémenté plusieurs fonctions dans cette classe TimerPerso, qui permettent de gérer le timer :
Ainsi, lors d'un appui sur le bouton *stop*, nous changeons l'état de la variable *pause* et lancons une interruption sur le thread du timer. Cela a pour effet de stopper le thread le plus rapidement possible.
L'appui sur le bouton stop ne tue pas le thread, mais empêche l'exécution de la fonction *iteration*, qui est normalement exécutée à interval régulier.
Pour quitter complètement le thread, on a la possibilité d'appeler la fonction *public void quit() ;* que nous avons implémenté. Lors de l'appui sur le bouton start, on change à nouveau la variable *pause* et on lance une interruption, pour reprendre l'éxecution du timer le plus rapidement possible.

Avec cette manière de faire, on est assuré de minimiser le temps entre l'appui sur un bouton et le lancement de l'action associée.
D'autre part, le thread etant endormi entre deux exécutions de la fonction *iteration*, on utilise un minimum de ressources CPU, comme le montre la figure 3.

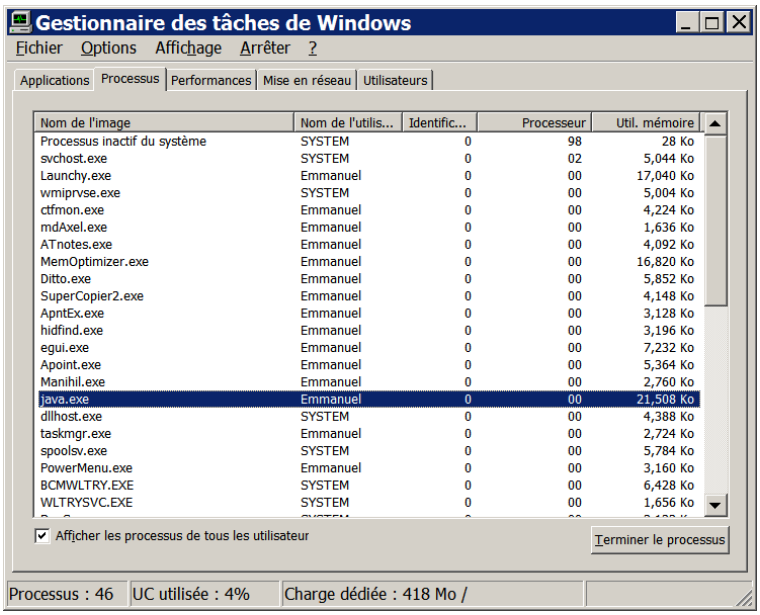


FIG. 3 – O pourcent d'utilisation du CPU lors de l'exécution du jeu (qu'il soit lancé ou en pause)

II La version finale du jeu

Voici à présent la dernière version du jeu, qui contient en plus un label pour l'affichage des scores, une hauteur aléatoire pour le caractère, et un label pour l'affichage du caractère entré au clavier.
De part le mode de gestion que nous avons choisi, la seule variable partagée entre les threads EventDispatchThread et TimerPerso est la variable *positionCaract*, qui stocke la position en X du caractère à l'écran.
En effet, la fonction *iteration*, qui est appelée par le timer, contient le code suivant :

```
public void iteration()
positionCaract+=5 ;
repaint() ;
```

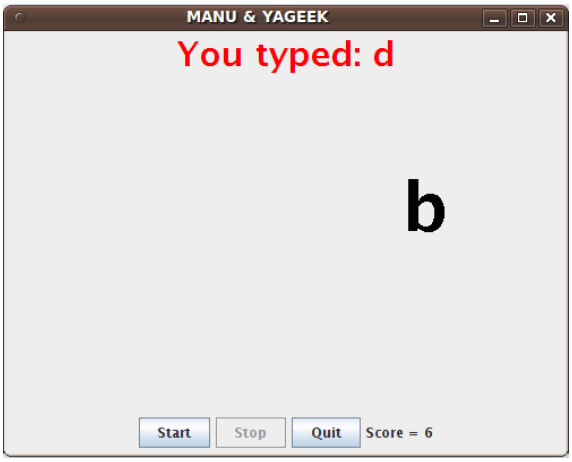


FIG. 4 – La version finale du jeu

Or cette ressource partagée n'implique pas de problèmes, comme nous l'avons expliqué dans la partie *Générateur aléatoire de caractères*.
L'ajout d'un label pour l'affichage des scores ne rajoute pas de ressources partagées dans notre cas, puisque sa mise à jour est gérée entièrement dans l'EventDispatchThread.

Un autre point qu'on peut mentionner est que le timer, en plus d'incrémenter la variable positionCaract, appelle la fonction repaint() pour mettre à jour l'affichage sur le JPanel.
Comme cette fonction est appelée depuis le thread du timer, on peut penser qu'elle va s'exécuter en parallèle. Mais ce n'est pas le cas, car la mise à jour de l'affichage (donc l'appel de la fonction paintComponent du JPanel) sera ajoutée à la queue d'évenements et faite séquentiellement par l'EventDispatchThread, comme le mentionne la javadoc de la fonction repaint().

On peut donc affirmer que notre programme est Thread-safe.

III Documentation

A Système de gestion de versions

Durant la réalisation de ce projet, nous avons utilisé un système de gestion de versions. Il permet de stocker les différentes versions du programme, et de visualiser facilement les différences.
Nous vons choisi un système SVN. Il est accessible librement en lecture.
L'adresse du SVN : `http ://code.google.com/p/jeudecaractere/`

B Licence du programme

Le projet est sous licence GNU GPL 2 (General Public License).