UKO

# Mini-Projet Java

*USB Memory Stick*

http://www.ukonline.be/programmation/java/

## ◇ Description

Dans ce mini-projet, on va écrire un *simulateur de stick USB*. Partant d'une interface décrivant un appareil de stockage de données, on va tout d'abord implémenter une classe représentant un stick USB et ensuite on va écrire une interface graphique (figure 1) permettant de voir précisément le contenu interne du stick USB et d'ajouter et supprimer des fichiers.

On va prendre un modèle de mémoire simplifié dans le cadre de ce mini-projet. La mémoire d'un appareil de stockage est divisée en *blocs* adjacents. Certains de ces blocs contiennent les données de fichiers tandis que d'autres blocs ne contiennent rien, ce sont des blocs vides.

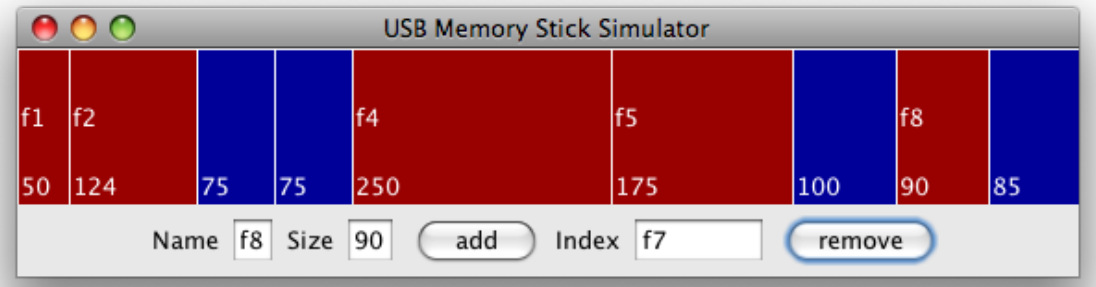



**Figure 1.** Exécution du programme `USBSimulator`

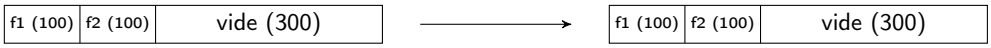
Lorsqu'il faut ajouter un fichier sur l'appareil de stockage, trois situations différentes peuvent se produire :

1. Il y a un bloc vide assez grand pour contenir le fichier. Ce bloc est divisé en deux parties : le fichier se place dans la première tandis que la seconde est un bloc vide plus petit. Voici par exemple comment se passe l'ajout d'un fichier `f1` de taille 200 :

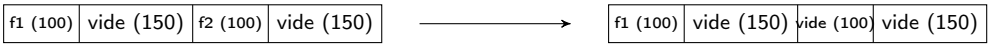

2. Il n'y a pas de bloc vide assez grand mais il reste assez de place en tout sur la mémoire. Elle doit être *défragmentée*, c'est-à-dire que tous les blocs vides sont rassemblés. Supposons que l'on souhaite ajouter un fichier `f3` de taille 200. Il n'y a pas de bloc vide assez grand, il faut donc d'abord défragmenter la mémoire :


Une fois la mémoire défragmentée, le fichier peut maintenant être ajouté sur la mémoire :

3. Dernière possibilité : il n'y a pas assez de place sur la mémoire, même si on l'a défragmentait. Dans ce cas, la mémoire n'est pas défragmentée et le fichier n'est pas ajouté. Supposons que l'on souhaite ajouter un fichier `f3` de taille 350 :




On doit également pouvoir supprimer un fichier de la mémoire. Le bloc contenant le fichier devient tout simplement un bloc vide. Voici un exemple où l'on supprime le fichier `f2` :




Vous remarquerez que la suppression du fichier `f2` fait qu'il y a maintenant trois blocs vides consécutifs. Ceux-ci ne vont pas être fusionnés en un bloc vide plus grand (ce serait bien entendu une optimisation possible mais qu'on ne va pas implémenter dans le cadre de ce mini-projet).

## ◇ Structure du programme

La figure 2 montre les différentes classes et interfaces utilisées dans ce projet. La classe `USBMemoryStick` est celle qui représente le stick USB et que vous devez implémenter. Elle utilise une classe interne `EmptyBlock` qui représente un bloc vide. Les deux classes suivantes sont des programmes. La classe `TestUSBMemoryStick` teste la classe `USBMemoryStick`. La classe `USBSimulator` créé une interface graphique qui permet de manipuler un stick USB.

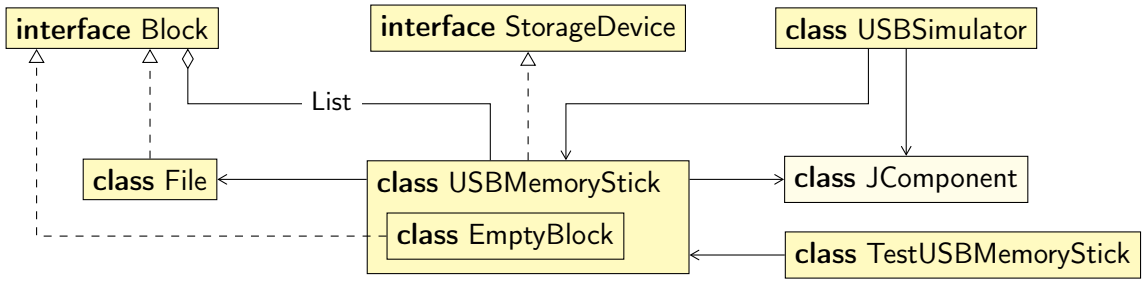



**Figure 2.** Diagramme de classe du mini-projet

## ◇ Implémentation

1. Commençons en douceur en implémentant la classe `File`. Un fichier est un `Block` et est identifié par un nom et un contenu (tous les deux sont des `String`). La longueur d'un fichier est le nombre de caractères de son contenu. Voici le squelette de la classe :

```
1  public final class File implements Block
2  {
3     private final String name, data;
4
5     /**
6      * @pre name, data != null, ! name.equals ("")
7      * @post An instance of this is created representing a file
8      *       with specified name and data
9      *       the size of this file is the number of characters of data
10     */
11    public File (String name, String data) { /* À compléter */ }
12 }
```

Pensez également définir une méthode `getName` qui permet d'obtenir le nom du fichier et à redéfinir la méthode `toString`.

2. Implémentez maintenant la classe `USBMemoryStick`. Celle-ci implémente l'interface `StorageDevice`. La mémoire est représente comme une `List` de `Block`. On va aussi utiliser deux variables d'instance pour la capacité (`capacity`) et l'espace utilisé sur le stick USB (`used`). N'oubliez pas non plus de définir la classe interne `EmptyBlock` représentant un bloc vide d'une certaine taille.

```
1  public final class USBMemoryStick implements StorageDevice
2  {
3     private final List<Block> blocks;
4     private final int capacity;
5     private int used;
6
7     /**
8      * @pre capacity > 0
9      * @post An instance of this is created, representing an USB memory stick
10    *       with specified capacity, and containing nothing
11    */
12    public USBMemoryStick (int capacity) { /* À compléter */ }
13
14    // À compléter
15
16    private final static class EmptyBlock implements Block
17    {
18       private final int size;
19
20       /**
21        * @pre size > 0
22        * @post An instance of this is created, representing
23        *       an empty block with the specified size
24        */
25       public EmptyBlock (int size) { /* À compléter */ }
26    }
27 }
```

3. Une fois la classe `USBMemoryStick` implémentée, il faut la tester. Écrivez une classe `TestUSBMemoryStick` avec une méthode `main` qui crée un stick USB et utilise toutes les méthodes de la classe `USBMemoryStick`. Si ce n'est déjà fait, redéfinissez la méthode `toString` dans la classe `USBMemoryStick` qui affiche la liste des blocs (vides et fichiers) qui sont sur le stick.

4. La dernière partie du projet consiste à écrire une petite interface graphique qui permet de consulter les blocs d'un stick USB et de lui ajouter ou supprimer des fichiers. Pour cela, définissez la méthode suivante dans la classe `USBMemoryStick`. Testez-là ensuite avec la classe `USBMemory` qui vous est donnée dans l'archive accompagnant ce projet.

```
1  public static JComponent createComponent (final USBMemoryStick usb)
2  {
3     return new JComponent()
4     {
5        @Override
6        protected void paintComponent (Graphics graphics)
7        {
8           super.paintComponent (graphics);
9           // À compléter ...
10       }
11    };
12 }
```

## ◇ Annexes

```java
// Block.java

/**
 * Interface representing a block
 *
 * @author Sébastien Combéfis
 * @version 2009-03-12
 */
public interface Block
{
   /**
    * Get the size of the block
    *
    * @pre -
    * @post The returned value contains the size of this block
    */
   public int getSize();
}
```

```java
// StorageDevice.java

/**
 * Interface representing a storage device
 *
 * @author Sébastien Combéfis
 * @version 2009-03-12
 */
public interface StorageDevice
{
   /**
    * Get the capacity of the device
    *
    * @pre -
    * @post The returned value contains the total capacity of this device
    */
   public int getCapacity();

   /**
    * Get the used space on the device
    *
    * @pre -
    * @post The returned value contains the space used on this device
    */
   public int getUsed();

   /**
    * Test whether the device is full
    *
    * @pre -
    * @post The returned value contains true if this device is true
    *       and false otherwise
    */
   public boolean isFull();
```

```
    /**
     * Add a file to the device
     *
     * @pre f != null
     * @post The file is added to the storage device if there is enough place
     * @throws FullException otherwise
     */
    public void addFile (File f) throws FullException;

    /**
     * Get a file from the device
     *
     * @pre name != null && ! name.equals ("")
     * @post The returned value contains a file with specified name
     *       if such file exists
     * @throws FileNotFoundException otherwise
     */
    public File getFile (String name) throws FileNotFoundException;

    /**
     * Remove a file from the device
     *
     * @pre name != null && ! name.equals ("")
     * @post The returned value contains a file with specified name
     *       if such file exists and this file is removed from the device
     * @throws FileNotFoundException otherwise
     */
    public File remove (String name)  throws FileNotFoundException;
}
```

```java
// FullException.java

/**
 * Exception throwned when trying to add a file to a full storage device
 *
 * @author Sébastien Combéfis
 * @version 2009-03-12
 */
@SuppressWarnings ("serial")
public final class FullException extends RuntimeException
{
   /**
    * Constructor
    *
    * @pre -
    * @post An instance of this is created
    */
   public FullException (String msg)
   {
      super (msg);
   }
}
```

```java
// FileNotFoundException.java

/**
 * Exception throwned when trying to access a non-existing file on a storage
 device
 *
 * @author Sébastien Combéfis
 * @version 2009-03-12
 */
@SuppressWarnings ("serial")
public final class FileNotFoundException extends Exception{}
```