Plugins

(les fichiers mentionnés dans ce document sont disponibles sur le portail)

Préliminaire

L'objectif de ce projet est la mise en place *progressive* d'une application qui s'adapte dynamiquement en fonction de "plugins" installés dans un répertoire.
Ce projet sera l'occasion de voir la mise en place du design-pattern *observer/observable* et d'utiliser certains aspects de la réflexivité.

Un exemple de l'application que vous allez réaliser au cours de ce projet est disponible. Pour cela récupérez l'archive `fichiers-plugins.zip` disponible sur le portail et décompressez la dans votre espace de travail. Vous devez avoir maintenant un répertoire contenant les dossiers `classes`, `available` et `repository`. Placez vous dans le dossier `plugins` et éxécutez la commande :

```
java -classpath classes:repository plugin.editor.Main &
```

La fenêtre qui apparaît contient une barre de menu. L'item du menu Tools propose un certain nombre de fonctionnalités (une pour l'instant). Ces outils correspondent à des opérations qui s'appliquent au texte contenu dans la zone de texte constituant la partie inférieure de l'application.
Vous pouvez expérimenter cette application en appliquant l'outil proposé sur un texte (que vous saisissez au clavier ou chargez depuis le menu File).

Cette application a la particularité qu'il est possible de l'enrichir en lui ajoutant dynamiquement, c'est-à-dire pendant qu'elle est en cours d'exécution, de nouveaux outils. Ce sont ces outils qui vont constituer les plugins évoqués ci-dessus.
Un certain nombre de plugins sont proposés, ils se trouvent dans le dossier `available`.
Copiez un par un les `.class` situés dans ce répertoire vers `repository`, observez ce qui se passe entre chaque copie dans le menu Tools et testez chacun des nouveaux outils proposés. Vous pouvez placer dans le même dossier un autre fichier (y compris d'extension `.class`) quelconque et, normalement, rien ne se passe au niveau de l'application. Pour être reconnu comme étant un plugin un fichier doit en effet satisfaire un certain nombre de contraintes.
Il est également possible de supprimer un outil, il vous suffit pour cela d'effacer le fichier `.class` correspondant du répertoire `repository`.

Le projet

L'objectif de ce projet sera l'implémentation de cette application.
Nous allons procéder progressivement. **Vous devez valider chaque étape (chaque exercice) avant de passer à la suivante.**

Des extraits de documentation sont fournis en annexe mais ils ne doivent pas dispenser de consulter les documentations complètes des ces classes ou méthodes.

Exercice 1 : Les timers
La classe `javax.swing.Timer` permet de définir des objets exécutant régulièrement une certaine tâche[1]. Son constructeur prend comme paramètre un `delay` (`int`) et un `ActionListener` dont la méthode `actionPerformed` est déclenchée tous les `delay` millisecondes une fois que le timer a été démarré grâce à la méthode `start()`.

Q 1 . Lisez la javadoc de `javax.swing.Timer`.

Q 2 . Pour vérifier votre compréhension de cette classe, écrivez un programme qui crée puis démarre un timer dont la fonction est d'afficher sur la sortie standard la date courante (`java.util.Calendar.getInstance().getTime()`) toutes les secondes.

Vous créerez en classe interne le type d'`ActionListener` dont vous aurez besoin.

(Projet) Expérimentez. N'oubliez pas de démarrer le timer ! Et pour éviter que l'application ne se termine aussitôt en stoppant le timer à peine démarré, placez un superbe `while (true) ;` à la fin de votre `main`.

[1] La classe `java.util.Timer` aurait aussi pu être utilisée mais ne les confondez pas.

Exercice 2 : Lister le contenu d'un répertoire
La classe `File` permet de créer des objets qui "représentent" des fichiers mais aussi un répertoire.

Q 1 . Lisez la javadoc de la méthode `public String[] list(FilenameFilter filter)` de la classe `java.io.File`.

Cette méthode a pour résultat le tableau des noms des fichiers acceptés (`accept()`) par l'objet de type `FilenameFilter` passé en paramètre.

Q 2 . Lisez la javadoc de la méthode `accept` de l'interface `java.io.FilenameFilter`

Q 3 . En vous appuyant sur l'écriture de deux classes qui implémentent cette interface, écrivez une classe dont les instances sont paramétrées par un répertoire dont le nom (son chemin) est donné à la création et disposent de deux méthodes :

1. l'une pour retourner la liste des noms de tous les fichiers de ce répertoire dont le nom commence par un `C`
2. l'autre pour retourner la liste des noms de tous les fichiers de ce répertoire dont l'extension est `.class`

Exercice 3 : Un vérificateur de nouveaux fichiers `.class`
Nous allons maintenant nous intéresser à la mise en place d'un mécanisme événementiel pour gérer la détection de l'apparition de fichiers dans un répertoire choisi.
Pour pouvoir faire cela nous allons mettre en œuvre le design-pattern *Observer*. Il s'agira d'émettre un évènement à chaque fois que l'on détecte qu'un nouveau fichier est ajouté dans le répertoire spécifié. Les objets "client" qui seront abonnés à ces évènements seront notifiés de cet événement et pourront réagir en conséquence.
L'objet émetteur de l'évènement, que nous appellerons `FileChecker`, examinera régulièrement le contenu du répertoire concerné à l'aide d'un objet `Timer`. On souhaite qu'en l'occurrence l'`ActionListener` du timer examine le contenu du répertoire voulu et informe de l'apparition d'un **nouveau** fichier `.class`. Pour chaque nouveau fichier détecté on créera et propagera un évènement (de type `FileEvent` dans la suite) pour notifier les abonnés (les "*observers*").

On applique ici une démarche méthodique et progressive pour la mise en place design pattern Observer/Observable qui suit les étapes suivantes :

1. définir la classe d'événements ;
2. définir l'interface des *listeners*, qui représente le type abstrait des *observers* ;
3. définir la classe émettrice (génératrice) des événements, qui correspond aux *observables* ;
4. définir les classes réceptrices (les *listeners*), qui représentent les *observers* concrets ;

Q 1 . Créez une classe `FileEvent` qui correspondra à l'évènement émis lorsqu'un nouveau fichier est ajouté.

Le nom du fichier concerné par l'événement, c'est-à-dire le fichier ajouté, est passé en paramètre du constructeur de l'objet évènement et mémorisé.

Q 2 . Créez l'interface `FileListener` associée à de tels évènements et définissant le type des obervateurs pour ces événements. Cette interface imposera la méthode `fileAdded`.

Q 3 . Créez une classe `FileChecker` qui sera la classe des objets émetteurs des évènements `FileEvent`.

Son comportement est celui décrit précédemment. Le constructeur de cette classe prendra en paramètre un objet `FilenameFilter` qui permet de définir les conditions pour filtrer les fichiers à surveiller, ainsi que le chemin du répertoire surveillé.

Il faut donc dans cette classe :

Q 3.1. gérer les objets de type `FileListener` qui pourront être les observateurs de cette classe.

Il faut pour cela ajouter le couple de méthodes permettant l'abonnement et le désabonnement de ces *listeners*.

Q 3.2. coder une méthode `fireFileAdded` qui crée et propage l'événement à tous les *listeners* abonnés. Le nom du fichier pourra être passé en paramètre de cette méthode.

Q 3.3. mettre en place le comportement qui se base sur un `timer` et un `FilenameFilter` pour détecter l'apparition des nouveaux fichiers recherchés dans le dossier surveillé et lorsque c'est le cas déclencher la propagation d'un évènement d'ajout grâce à un appel à `fireFileAdded(...)`.

Pour éviter de détecter deux fois le même fichier, il sera nécessaire de maintenir la liste des fichiers "déjà connus" par le `FileChecker`.

Q 4 . Proposez des tests permettant de vérifier le bon comportement de `FileChecker`.

Q 5 . Créez une classe implémentant `FileListener` et dont la seule réaction à l'événement d'ajout de fichier est d'afficher le message "`nouveau .class :` *xxxx* `détecté`" pour chaque évènement émis.

Q 6 . (Projet) Expérimentez l'ensemble. N'oubliez pas d'abonner le `FileListener` au `FileChecker` ni de démarrer le timer du `FileChecker` (à nouveau un `while (true);` sera nécessaire).

Q 7 . On souhaite maintenant détecter également la suppression de fichiers. Faites les modifications nécessaires :

- dans `FileListener` pour ajouter une méthode `fileRemoved`
- dans `FileChecker` pour détecter la disparition d'un fichier "connu" et propager l'événement à l'aide d'une méthode `fireFileRemoved(...)`
- dans la classe implémentant `FileListener` pour prendre en compte la modification de l'interface, on affichera dans ce nouveau cas "`.class` *xxxx* `supprimé détecté`".

Exercice 4 : Première prise en compte des plugins

Nous allons maintenant nous attaquer à l'application initialement annoncée en étendant ce qui a déjà été réalisé.
Commençons par poser certaines contraintes pour définir ce qui sera considéré comme un plugin. Nous allons ainsi supposer que les classes candidates doivent :

- implémenter l'interface `plugin.Plugin` (éventuellement indirectement - cf. `isAssignableFrom` dans la classe `Class`)

```
package plugin;
public interface Plugin {
  public String transform(String s) ;
  public String getLabel() ;
  public String helpMessage() ;
}
```

- appartenir au paquetage `plugins`.
- fournir un constructeur sans paramètre (cf. `getConstructor()` dans `Class`).

Q 1 . Créez une nouvelle classe `PluginFilter` implémentant `FilenameFilter` qui n'accepte que les fichiers correspondant à un plugin. Il faut donc :

1. avoir un `.class`,
2. charger l'instance `Class` associée grâce à `Class.forName(...)` (attention à supprimer l'extension du nom du fichier),
3. vérifier que c'est une classe qui satisfait les conditions mentionnées ci-dessus.

Q 2 . Proposez des tests pour `PluginFilter`.

Q 3 . Ecrivez une classe `SimplePluginObserver` qui affiche un message d'information pour chaque fichier *plugin* ajouté ou retiré du répertoire `extensions`.

Q 4 . (Projet) Expérimentez votre `SimplePluginObserver` en l'abonnant à un `FileChecker` de l'exercice précédent qui utilise une instance de `PluginFilter` pour filtrer les fichiers.

NB : *Il faut que le répertoire contenant les* `.class` *des plugins soit dans le classpath pour permettre au* `forName` *de bien fonctionner. C'est ce qui est fait dans la ligne de commande d'exécution avec l'option* `-classpath classes:repository` *de l'application fournie en tout début du sujet.*
Si besoin sous ECLIPSE *vous pouvez adapter le classpath d'exécution via le 4ème onglet du menu de paramétrage de Run...). Sous Eclipse, vous pouvez également adapter le répertoire de travail du programme dans le 4ème onglet du menu de paramétrage de Run...) (tout en bas).*

Exercice 5 : L'application graphique

On va maintenant créer l'application demandée complète, c'est-à-dire avec l'interface graphique.
Outre le fait d'utiliser d'un `FileChecker` qui utilise le `PluginFilter` précédent pour la détection des nouveaux fichiers, l'application se base sur un nouveau type d'*observer*, implémentant donc `FileListener`. Il faudra définir celui-ci pour qu'il gère l'ajout ou le retrait d'éléments de menu lors de l'apparition ou de la disparition de fichiers de plugin, c'est-à-dire lorsqu'il est notifié de l'un ou l'autre des événements.
Il s'agit en fait d'ajouter (ou de supprimer) des `JMenuItem` dans un `JMenu` défini. Le label de cet item est le `getLabel` du plugin et l'action sur ce `JMenuItem` consiste à appeler la méthode `execute` du plugin. Il faudra donc construire une instance de la classe de plugin détectée (voir `getConstructor()` dans `Class` puis `newInstance()` dans `Contructor`) et la lier au `JMenuItem`. Il sera certainement pertinent de définir une classe de `JMenuItem` spécifique.

Q 1 . (Projet) Réalisez l'application avec l'interface graphique.

Pour la zone de texte vous pouvez utiliser un `JScrollPane`. Si vous souhaitez le mettre en place, pour le choix d'un fichier texte à afficher dans l'éditeur vous pouvez utiliser un objet `JFileChooser`.

A vous d'imaginer d'autres plugins que ceux suggérés (codage de César, suppression des voyelles, ajout d'une signature en fin de texte, etc.).

Projet

Vous devez produire les différents programmes indiqués dans les exercices précédents. Vous indiquerez clairement dans votre fichier *readme* comment exécuter ces différents programmes.
Comme déjà indiqué précédemment, vous ne devez pas passer à l'exercice suivant tant que la version de l'exercice en cours n'est pas totalement opérationnelle.

Annexe : documentation

```
================================================================================================
javx.swing.Timer
--------------------
public Timer(int delay, ActionListener listener)

Creates a Timer and initializes both the initial delay and between-event delay to delay
milliseconds. If delay is less than or equal to zero, the timer fires as soon as it is
started. If listener is not null, it's registered as an action listener on the timer.

  Parameters:
    delay - milliseconds for the initial and between-event delay listener - an initial
  listener; can be null

--------------------
public void start()

  Starts the Timer, causing it to start sending action events to its
  listeners.

================================================================================================
java.util.File
--------------------
public String[] list(FilenameFilter filter)

Returns an array of strings naming the files and directories in the directory denoted by
this abstract pathname that satisfy the specified filter. The behavior of this method is
the same as that of the list() method, except that the strings in the returned array must
satisfy the filter. If the given filter is null then all names are accepted. Otherwise, a
name satisfies the filter if and only if the value true results when the
FilenameFilter.accept(File, String) method of the filter is invoked on this abstract
pathname and the name of a file or directory in the directory that it denotes.

  Parameters:
    filter - A filename filter
  Returns:
    An array of strings naming the files and directories in the directory denoted by this
  abstract pathname that were accepted by the given filter. The array will be empty if the
  directory is empty or if no names were accepted by the filter. Returns null if this
  abstract pathname does not denote a directory, or if an I/O error occurs.
  Throws:
    * SecurityException - If a security manager exists and its
    * SecurityManager.checkRead(String) method denies read access to the directory

================================================================================================
java.io.FileNameFilter
--------------------
public boolean accept(File dir, String name)

Tests if a specified file should be included in a file list.

    Parameters:
      dir - the directory in which the file was found.
      name - the name of the file.
    Returns:
      true if and only if the name should be included in the file list;
```

    false otherwise.

==
java.lang.Class
----
public boolean isAssignableFrom(Class<?> cls)

Determines if the class or interface represented by this Class object is either the same as, or is a superclass or superinterface of, the class or interface represented by the specified Class parameter. It returns true if so; otherwise it returns false. If this Class object represents a primitive type, this method returns true if the specified Class parameter is exactly this Class object; otherwise it returns false.

Specifically, this method tests whether the type represented by the specified Class parameter can be converted to the type represented by this Class object via an identity conversion or via a widening reference conversion. See The Java Language Specification, sections 5.1.1 and 5.1.4 , for details.

  Parameters:
      cls - the Class object to be checked
  Returns:
      the boolean value indicating whether objects of the type cls can be
      assigned to objects of this class
  Throws:
      NullPointerException - if the specified Class parameter is null.

----
public static Class<?> forName(String className) throws ClassNotFoundException

Returns the Class object associated with the class or interface with the given string name. Invoking this method is equivalent to:

      Class.forName(className, true, currentLoader)

 where currentLoader denotes the defining class loader of the current class.

 For example, the following code fragment returns the runtime Class descriptor for the class named java.lang.Thread:

      Class t = Class.forName("java.lang.Thread")

  A call to forName("X") causes the class named X to be initialized.

  Parameters:
    className - the fully qualified name of the desired class.
  Returns:
    the Class object for the class with the specified name.
  Throws:
    * LinkageError - if the linkage fails
    * ExceptionInInitializerError - if the initialization provoked by this method fails
    * ClassNotFoundException - if the class cannot be located

----
public Constructor<T> getConstructor(Class<?>... parameterTypes)
        throws NoSuchMethodException, SecurityException

Returns a Constructor object that reflects the specified public constructor of the class represented by this Class object. The parameterTypes parameter is an array of Class objects that identify the constructor's formal parameter types, in declared order. If this Class object represents an inner class declared in a non-static context, the formal parameter types include the explicit enclosing instance as the first parameter.

The constructor to reflect is the public constructor of the class represented by this

```
Class object whose formal parameter types match those specified by parameterTypes.

Parameters:
    parameterTypes - the parameter array
Returns:
    the Constructor object of the public constructor that matches the specified parameterTypes
Throws:
    * NoSuchMethodException - if a matching method is not found.

    * SecurityException - If a security manager, s, is present and any of the following
    conditions is met:

        -  invocation of s.checkMemberAccess(this, Member.PUBLIC) denies access to the constructor
        - the caller's class loader is not the same as or an ancestor of the class
        loader for the current class and invocation of s.checkPackageAccess() denies
        access to the package of this class

================================================================================
java.lang.reflect.Contrsuctor
--------------------
public T newInstanceObject... initargs) throws InstantiationException, IllegalAccessException,
              IllegalArgumentException,InvocationTargetException

Uses the constructor represented by this Constructor object to create and initialize a
new instance of the constructor's declaring class, with the specified initialization
parameters. Individual parameters are automatically unwrapped to match primitive formal
parameters, and both primitive and reference parameters are subject to method
invocation conversions as necessary.

If the number of formal parameters required by the underlying constructor is 0, the
supplied initargs array may be of length 0 or null.

(...)

If the required access and argument checks succeed and the instantiation will proceed,
the constructor's declaring class is initialized if it has not already been
initialized.

If the constructor completes normally, returns the newly created and initialized instance.

Parameters:

initargs - array of objects to be passed as arguments to the constructor call; values
of primitive types are wrapped in a wrapper object of the appropriate type (e.g. a
float in a Float)

Returns:
    a new object created by calling the constructor this object represents
Throws:
     * IllegalAccessException - if this Constructor object is enforcing Java language access
     control and the underlying constructor is inaccessible.
     * IllegalArgumentException - if the number of actual and formal parameters differ;
     if an unwrapping conversion for primitive arguments fails; or if, after possible
     unwrapping, a parameter value cannot be converted to the corresponding formal
     parameter type by a method invocation conversion; if this constructor pertains to
     an enum type.
     * InstantiationException - if the class that declares the underlying constructor
     represents an abstract class.
     * InvocationTargetException - if the underlying constructor throws an exception.
     * ExceptionInInitializerError - if the initialization provoked by this method
     fails.
================================================================================
```