Programmation Orientée Objet
M2 CCI 2008–2009
hoenen@icps.u-strasbg.fr

Université de Strasbourg
Département d'informatique
4 février 2009

TP6 : IDE, javadoc, JAR

1 Eclipse

Le but de cette partie est d'avoir un premier contact avec un IDE (*Integrated Development Environment*) pour programmer en Java. Nous prendrons comme exemple l'IDE libre Eclipse.

Lancement d'Eclipse

Ajoutez en premier lieu un répertoire `projets` dans lequel on placera tous les fichiers développés dans Eclipse. Lancez Eclipse par la commande `eclipse-3.2` sur Turing (le démarrage de l'application peut prendre du temps). Si l'application demande de sélectionner un *workspace*, choisissez le répertoire que vous venez de créer.

Premier projet

Eclipse fonctionne par *projet*. Dans un premier temps, créez un projet par *File→New→Project...*, puis sélectionnez *Java Project* et *next*. Donnez alors un nom au projet (par exemple `premier projet`) puis *finish*. Le projet apparait normalement dans une fenêtre sur la gauche. On peut alors ajouter un package dans ce projet en faisant un clic droit sur le projet puis en sélectionnant *New→Package* : rentrez un nom de package (par exemple `tp6`) puis *finish*. De même, un clic droit sur le package suivit de *New→Class* permet de créer une classe dans ce package : il existe alors plusieurs options, notamment sur les modificateurs et un ensemble de méthodes à créer automatiquement.

Créez la classe publique `HelloWorld` qui contient une méthode `main`. La fenêtre centrale donne le squelette de cette classe, en précisant le package, le nom de la classe et le profile de la méthode `main` précédée par des commentaires javadoc. Ajoutez `System.out.println(‘‘coucou depuis eclipse’’);` dans le corps du `main` : vous remarquez que Eclipse vous propose au fur et à mesure les différentes méthodes utilisables depuis une classe donnée.

Pour lancer un programme dans Eclipse, depuis la classe contenant le main, allez dans *Run→Run As→Java Application*. Ensuite, retournez dans le même menu pour exécuter le programme dans un onglet Console situé en bas de la fenêtre, ou clickez sur l'icône run (rond vert avec une flèche blanche). Si le programme nécessite des arguments, ceux-ci peuvent être précisés dans *Run→Run...→Arguments* dans la partie Program arguments.

Héritage, etc...

Créez dans le package `tp6` une interface `MonInterface`. Cette interface définit le profil d'une méthode `void afficher()`. Créez également une classe `MaClasse` qui possède un attribut `protected String` et un constructeur qui initialise la valeur de cette chaîne de caractères. Enfin, ajoutez une classe `MaSousClasse` qui hérite de `MaClasse` et implémente `MonInterface` (cela se précise dans les options lors du choix du nom de la nouvelle classe). Cochez la case permettant d'ajouter automatiquement le constructeur de la super-classe. Ajoutez un attribut `private int n` et donnez le code de la méthode `afficher` (par exemple une boucle affichant un numéro de `0` à `n-1` suivi de la chaîne contenue dans la super-classe). L'attribut `n` n'étant pas initialisé dans le constructeur et étant déclaré `private`, ajoutez des méthodes *get* et *set* pour mettre à jour et lire sa valeur. Cela peut se faire de manière automatique en faisant un clic droit sur le code, puis en sélectionnant *Source→Generate Getters and Setters...* (vous pouvez alors choisir quelles méthodes get et set implanter et où les placer).

Ensuite, Eclipse permet de répercuter aux sous-classes des changements effectués dans des super-classes et des interfaces. Pour cela, il faut utiliser le *refactoring* : sélectionnez la méthode `afficher` de

l'interface `MonInterface`, puis faites un clic droit pour accéder à *Refactor→Change Method Signature*. Changez le retour de la méthode de `void` en `String`. Ce changement, effectué par refactor (vous pouvez également changer le nom ou le modificateur), est répercuté dans toutes les classes qui implémentent cette interface. Vous remarquerez que du coup, la classe `MaSousClasse` provoque une erreur (signalée par un pictogramme rouge) étant donné que le code de la méthode `afficher` ne possède pas d'instruction `return`. Ajoutez-la.

2 javadoc

Nous avons déjà vu que les commentaires sous la forme

```
/**
 * ceci est un commentaire javadoc
 */
```

sont utilisés pour générer une documentation automatique des classes grâce à l'outil *javadoc*. La page web http://java.sun.com/j2se/javadoc/writingdoccomments donne de nombreux détails sur la manière dont ces commentaires doivent être utilisés. Pour faire simple, un commentaire javadoc comporte deux parties : un texte de description de la classe, méthode ou interface, et une suite de *tags* comme par exemple `@param` (paramètres d'entrée), `@return` (valeur de retour), `@author` (nom du programmeur), `@exception` (type d'exception lancée et condition de lancement), `@deprecated` (si la méthode n'est plus à utiliser car une nouvelle méthode existe), etc... On peut également formater le texte en utilisant des balises html, étant donné que la documentation sera générée sous la forme de page html dans le style de l'API officielle Java.

Généralement un commentaire javadoc est organisé à peu près comme suit :

```
/**
 * Petit texte descriptif de la classe
 * @author l'auteur de la classe
 */
public class MaClasse {

  /**
   * Description de la méthode qui suit
   * @param nom du paramètre suivit de sa description
   * @return description de la valeur retournée par la fonction
   */
  public void MaMethode(int x) {
    ...
  }
}
```

Une fois le code source correctement commenté, la documentation html est générée en appelant l'outil `javadoc` suivit des fichiers `.java` dont les commentaires sont à prendre en compte dans la documentation. Il peut être pratique de ne pas mélanger les fichiers sources, les fichiers classe et les fichiers de documentation. Pour cela, ajoutez un répertoire `document` au même niveau que les répertoires `sources` et `classes`, puis générez la documentation de tous les fichiers contenus dans un répertoire par la commande `javadoc *.java -d $CLASSPATH/../document`.

3 JAR

Le JAR (Java ARchive) est un format d'archivage de fichier propre à Java. Une archive JAR permet de regrouper dans un même fichier un ensemble de fichiers `.class` (par exemple appartenant à un même package), de les signer numériquement pour en acroître la sécurité et l'intégrité, et de les compresser en

utilisant le format de compression ZIP. La commande `jar` permet de créer, lire, manipuler des archives au format JAR et s'utilise de la manière suivante :

- création d'une archive : `jar cf jar-file input-file(s)`
- voir le contenu d'une archive : `jar tf jar-file`
- extraire le contenu d'une archive : `jar xf jar-file`
- extraire une partie d'une archive : `jar xf jar-file archived-file(s)`
- exécuter une application contenue dans une archive : `java -jar app.jar`

Lorsqu'un JAR est créé, un manifeste est ajouté automatiquement dans `META-INF/MANIFEST.MF`. Ce manifeste permet notamment d'ajouter un point d'entrée dans le JAR : en définissant la classe contenant la méthode `main` principale, on rend le JAR exécutable. Pour ajouter ce point d'entrée, il faut utiliser la commande `jar cfm jar-file manifest-addition input-file(s)` lors de la création de l'archive JAR. Le fichier passé en paramètre à la place de `manifest-addition` (par exemple un fichier `Manifest.txt`) doit contenir une ligne `Main-Class: MyPackage.MyClass` où `MyPackage.MyClass` est la classe exécutable contenant le `main` du programme. Il faut prendre garde à ce que le fichier `Manifest.txt` se termine par un retour chariot. Par exemple, la commande `jar cfm MyJar.jar Manifest.txt MyPackage/*.class` créée un JAR exécutable qui peut ensuite être lancé directement par la commande `java -jar MyJar.jar` (ou en faisant un double clic dessus dans certains systèmes d'exploitation modernes).

Reprenez un exercice d'un TP précédent et générez une documentation javadoc dans un répertoire séparé. Regroupez dans un JAR les fichiers `.class` utilisés par une même application et rendez ce JAR exécutable en y ajoutant un manifeste.