Introspection

Introduction

Java est, par design, un langage statiquement typé : chaque variable possède un type écrit dans le code source même et qui permet de déterminer les opérations que l'on peut effectuer. On pourra par exemple appeler telle méthode, mais pas telle autre. Grâce à ça, Eclipse est capable de signaler **déjà à l'édition** les appels de méthodes fautifs, les accès aux attributs absents, etc.

L'inconvénient de cette approche est qu'elle limite les possibilités à l'exécution. Par exemple on ne pourra pas utiliser une classe inconnue que l'on vient de recevoir du réseau car on n'a pas de variable de ce type. Cette approche implique aussi des limitations dans l'écriture même du code : admettons qu'une méthode ait besoin de créer une instance, on ne pourra pas passer la classe de l'instance à créer en paramètre de la méthode.

```
public class StaticTypingAndFactory {

	public Object creeUnInstance(Class clss) {
		return new clss;
	}

}
```

A la place il faudra lui passer une factory qui emballe la mécanique de création de l'instance.

```
interface Factory {
	Object newInstance();
}

class StaticTypingAndFactory {

	public Object creeUnInstance(Factory factory) {
		return factory.newInstance();
	}

}
```

De même, on ne pourra pas appeler une méthode en lui donnant la référence d'une autre méthode qu'elle devrait appeler. De nouveau il faudra emballer cet appel d'une manière ou d'une autre.

Java fournit un mécanisme pour atteindre ce degré de flexibilité : l'introspection aussi appelé réflexivité. Ce mécanisme permet à un programme Java de regarder son propre code, de manipuler

les classes, méthodes et attributs à l'aide de types dédiés. On pourra ainsi avoir une variable qui référencie une classe et sera capable de générer des instances. Une autre variable référenciera une méthode de cette classe, que l'on pourra invoquer sur base de l'instance générée. Tout ceci se passe dynamiquement, à l'exécution même du programme.

Introspection d'une classe

C'est le type `Class` qui permet d'introspecter une classe ou une interface. On peut obtenir une instance de ce type de plusieurs manières :

- Sur base d'un nom de classe `XXXX`, en utilisant l'attribut `.class` : `XXXX.class`
- Sur base d'une instance de classe `XXXX`, en utilisant la méthode `getClass()` : new `XXXX().getClass()`
- Sur base du nom pleinement qualifié d'une classe, en utilisant la méthode statique `forName` de `Class` : `Class.forName("be.ipl.introspection.XXXX") ;`

Consultez la javadoc de `Class` pour en connaître toutes les possibilités. En particulier les méthodes suivantes sont intéressantes :

- `newInstance()` : retourne une nouvelle instance de la classe, initialisée par le constructeur sans paramètre.
- `getMethods()` : retourne un tableau de `Method`, chaque entrée représentant une méthode publique de la classe.
- `getDeclaredMethods()` : retourne un tableau de `Method`, mais en se limitant aux méthodes définies ou redéfinies par la classe.
- `getConstructors()` : retourne un tableau de `Constructor`, chaque entrée représentant un constructeur public de la classe. Ceci permettra d'instancier la classe en appelant un des constructeurs avec paramètre(s).
- `getFields()` : retourne un tableau de `Field`, chaque entrée représentant un attribut public de la classe.
- `getModifiers()` : retourne les modificateurs de la classe (`public`, `private`, …). La classe `Modifier` permet de tester les différents modificateurs.
- `getName()` : retourne le nom pleinement qualifié de la classe, sous forme de `String`.
- `getInterfaces()` : retourne un tableau de `Class`, chaque entrée représentant une interface implémentée par cette classe.
- `getSuperClass()` : retourne une instance de `Class` représentant la classe parent.

Introspection d'une méthode

C'est la classe `Method` qui permet d'introspecter une méthode. En général on obtient une instance de `Method` en demandant à une instance de Class. De nouveau consultez la Javadoc de `Method` pour en connaître toutes les possibilités. En particulier, les méthodes intéressantes sont :

- `invoke(Object obj, Object…. args)` : invoque cette méthode sur l'objet `obj` (si la méthode est statique, ce paramètre est ignoré), en lui passant les paramètres `args`. Le résultat de l'appel est retourné dans un `Object`. Si la méthode ne retourne rien (`void`), alors `l'Object` retourné est `null`. Les types primitifs sont automatiquement

emballés/déballés de leur type objet correspondant, aussi bien pour les paramètres que pour le type de retour. Si une exception s'échappe de la méthode, cette exception est emballée dans une `InvocationTargetException`. Si on n'a pas le droit d'appeler la méthode, une `IllegalAccessException` est levée. Si les paramètres fournis ne correspondent pas à ceux attendus, une `IllegalArgumentException` est levée.

- `getDeclaringClass()` : retourne la classe sur laquelle cette méthode est définie ou redéfinie en dernier.
- `getModifiers()` : retourne les modificateurs de la méthode (`public`, `private`, …). La classe `Modifier` permet de tester les différents modificateurs.
- `getName()` : retourne le nom de la méthode.
- `getParameterTypes()` : retourne un tableau listant dans l'ordre les `Class` des différents paramètres attendus par la méthode.
- `getReturnType()` : retourne la `Class` du type de retour de la méthode. Si la méthode ne renvoie rien, son type de retour sera de la `Class void.class`.

Introspection d'un attribut

La classe `Field` permet d'introspecter les attributs d'une classe. En général on obtient les attributs en demandant à une instance de Class. Comme toujours consultez la Javadoc de `Field` pour en connaître toutes les possibilités. En particulier les méthodes suivantes sont intéressantes :

- `get(Object obj)` : obtient la valeur de cet attribut de `obj`.
- `getName()` : retourne le nom de l'attribut.
- `getType()` : retourne une `Class` correspondante au type de cet attribut.
- `set(Object obj, Object value)` : change la valeur de cet attribut de `obj`.

Introspection et frameworks

Par l'introspection il devient possible d'écrire un bout de logiciel qui manipule un autre bout de logiciel. Le bout de logiciel qui manipule est appelé framework car il définit un cadre de fonctionnement pour l'autre bout de logiciel qui sera lui appelé le client. Le framework prend à sa charge la résolution d'une partie bien précise du problème général de l'application. Le client du framework est donc soulagé de cette partie, et peut se concentrer beaucoup plus sur les aspects spécifiques de l'application (en général le business).

En réalité on a des frameworks beaucoup plus complexes :

- Framework web qui sur base d'une requête web trouvera la ou les méthodes à appeler dans le programme.
- Framework de persistance qui pourra sauver le contenu d'une ou plusieurs instances dans un fichier ou une base de données pour le(s) récupérer plus tard.
- …

Pour accomplir leurs tâches correctement, ces frameworks ont souvent besoin d'informations complémentaires à l'introspection directe. L'exemple typique est la sécurité : l'administrateur d'un

site web devra être capable d'invoquer certaines méthodes non disponibles aux utilisateurs normaux. Il faut donc donner une information complémentaire aux frameworks pour leur permettre de prendre ces décisions adéquatement. C'est à cela que servent les annotations en Java.