

Introspection et réflexivité

Principes et application à Java

Philippe Collet

D'après un cours de Michel Buffa

Master 1 IFI

2014-2015

<http://deptinfo.unice.fr/twiki/bin/view/Minfo/GenieLog1415>

Plan

- RTTI en Java
 - Réflexivité : définitions
 - Analyser une classe
 - Analyser un objet
 - Exemple
 - Manipuler une méthode
 - Annotation
-
- Par défaut, les exemples sont basés sur Java 2 / JDK 1.4.x
 - Zoom sur les extensions amenées depuis Java 5 et 6

La classe Class

RTTI en java

Run-Time Type Identification

- Java maintient ce qu'on appelle l'Identification de Type au Run-Time (RTTI) sur tous les objets
- Permet de connaître la classe *réelle* d'un objet
- Permet d'implémenter la liaison dynamique (quelle méthode est réellement appelée) :

```
ObjetGraphique o = new Cercle();  
o.draw(); // de Cercle ou de ObjetGraphique ?
```

La classe `Class` et le RTTI

- Une classe permet d'accéder au RTTI, la classe `Class`

```
Personne p = new Personne("Maurice");
```

```
Personne p1 = new Professeur("Yoda")
```

```
Class cl = p.getClass();
```

```
System.out.println(cl.getName() + " " +  
    p.getNom());
```

affiche

```
"Personne Maurice"
```

```
"Professeur Yoda" avec p1...
```

Le suffixe .class

- On peut aussi obtenir un objet de type Class :

```
Class c11 = Professeur.class;
```

```
Class c12 = int.class;
```

```
Class c13 = double.class;
```

...

- Utile pour tester le type avec des types prédéfinies
- Un objet de type **Class** décrit un TYPE, pas forcément une CLASSE !

Nom = Class ?

- On peut obtenir une classe à partir de son nom (~ langages fonctionnels)

```
String nomClasse =  
    "com.wars.star.Professeur";  
Class cl = Class.forName(nomClasse);
```

- **nomClasse** peut être le nom d'une interface ou d'une classe
- Utile pour **charger des classes** dont on ne connaît pas le nom à l'avance

Créer des instances sans new

- Utilisation de la méthode `newInstance()` de la classe **Class**

```
String nomClasse = "Professeur";
```

```
Class c1 = Class.forName(nomClasse);
```

```
Object o = c1.newInstance();
```

- Note : il existe `newInstance(Object [] params)` de la classe `Constructor (java.lang.reflect)`

Méthodes de la classe Class (Java 2)

```
String getName();  
Class getSuperClass();  
Class [] getInterfaces();  
boolean isInterface();  
String toString();  
static Class forName(String name);  
Object newInstance();
```

Méthodes de la classe **Class<T>** (Java 5)

```
String getName();
```

```
TypeVariable<Class<T>>[] getTypeParameters();
```

```
Class<? super T> getSuperClass();
```

```
Class [] getInterfaces();
```

```
boolean isInterface();
```

```
String toString();
```

```
static Class<?> forName(String name);
```

```
T newInstance();
```

```
Constructor<T> getConstructor(Class...  
parameterTypes);
```

La réflexivité

Définition

- On appelle « réflexif » un système capable de se représenter lui-même
 - Pour les langages informatiques, la réflexivité s'exprime comme la capacité pour un langage à décrire les aspects considérés comme implicite dans le langage
 - Dans le cas d'un langage objet:
 - Accès à la classe d'un objet: les classes comme des objets
 - Instanciation de classes dont le nom est connu seulement à l'exécution
 - Accès aux attributs et aux méthodes des objets
 - Invocation de méthode (envoi de message) dont le nom n'est connu qu'à l'exécution

Schéma objet méta-circulaire



- Les classes sont des instances de Class et donc des objets
 - possèdent des attributs : les attributs des instances, leurs méthodes...
 - possèdent des méthodes (on peut donc leur envoyer des messages)
 - créer des instances
 - connaître la valeur de ses attributs
 - appeler des méthodes
- La classe Object est une classe et donc une instance de Class, elle peut donc
 - être manipulée de la même manière²

Réflexivité, introspection, méta-programmation ?

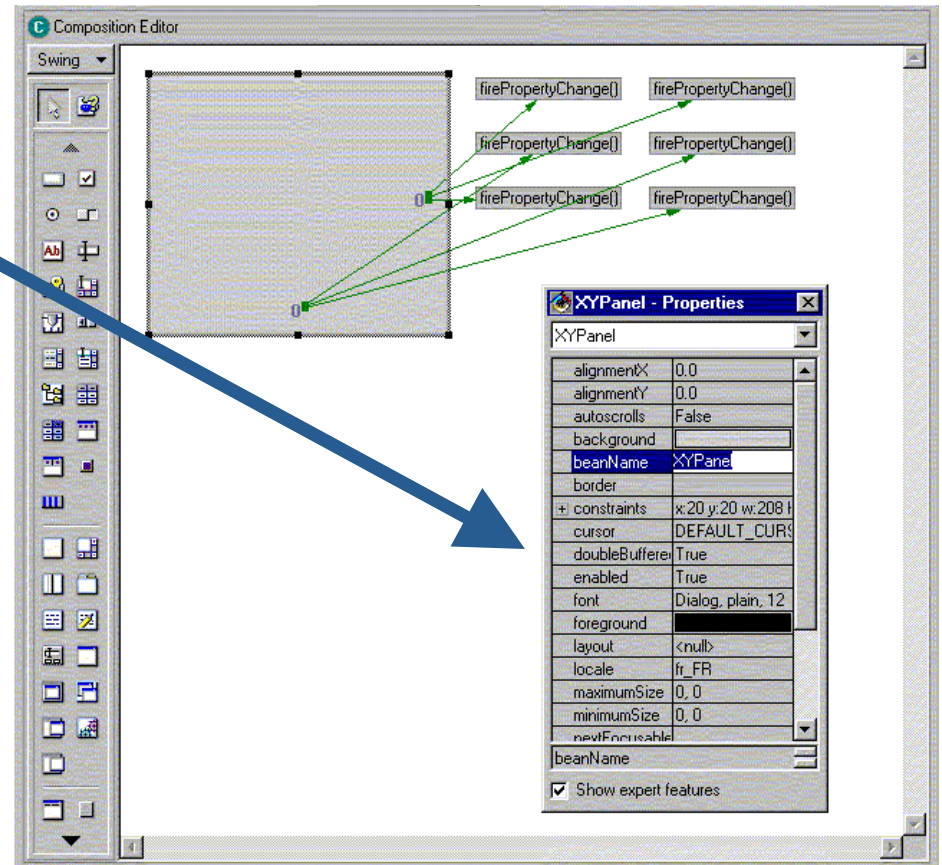
- Introspection
 - connaître/inspecter les classes, les objets, les méthodes à l'exécution
 - Le faire dans le même langage = introspection réflexive
- Meta-programmation
 - capacité de modifier les mécanismes du langage à l'aide d'un programme META
- Meta + réflex = inspecter et modifier le comportement
- Java = introspection réflexive, pas de protocole de méta-programmation

Comment faire en Java ?

- Classe **Class** minimale en Java 1.0
- Amélioration dès la version 1.1 à cause des Java Beans et des Objets Distribués (RMI...)
 - Outils (RAD en particulier : JBuilder, Visual Café, ...) ont besoin d'interroger à l'exécution des nouvelles classes (beans)
 - Des classes se déplacent dans les architectures distribuées
- Le package qui fournit ces possibilités est **java.lang.reflect**
 - Analyser des classes à l'exécution,
 - Inspecter des objets à l'exécution,
 - Ecrire une méthode toString() générique,
 - Manipuler des tableaux génériques,
 - Manipuler des méthodes comme des pointeurs sur fonction du C/C++...

Réflexivité dans un outil RAD

- Affichage des propriétés dans une fenêtre d'édition
- Propriété = attribut avec accesseur + modificateur (optionnel)
- Interrogation dynamique de la classe à la recherche de `get...()` et de `set...()`



Analyser une classe

Analyser une classe

- Trois classes dans `java.lang.reflect`
 - `Field`, `Method`, `Constructor`
 - Toutes possèdent `getName()` ;
 - `Field` possède `getType()` qui renvoie un objet de type `Class`.
 - `Method` et `Constructor` ont des méthodes pour obtenir le type de retour et le type des paramètres,

Analyser une classe (suite)

- Ces trois classes possèdent `getModifiers()` qui renvoie un `int`, dont les bits à 0 ou à 1 signifient `static`, `public`, `private`, etc...
 - On utilise les méthodes statiques de `java.lang.reflect.Modifier` pour utiliser cette valeur
 - On trouve dans `Modifier` les méthodes `toString(int)`, `isFinal(int)`, `isPublic(int)`, `isPrivate(int)`

Exemple d'utilisation

- Le programme du TP saura produire, à partir du nom d'une classe et d'un fichier .class

```
Class java.lang.double extends java.lang.number {  
    public static final double POSITIVE_INFINITY;  
    public static final double NEGATIVE_INFINITY;  
    public static final double NaN;  
  
    ...  
    public static java.lang.Double(double) ;  
    public static java.lang.Double(java.lang.String) ;  
  
    ...  
    public static java.lang.String toString(double) ;  
    public static boolean isNaN(double) ;  
    public boolean equals(java.lang.Object) ;  
  
    ...  
}
```

Méthodes de la classe Class

- **Field[] getFields()**
 - Ne renvoie **que les attributs public, locaux et hérités**
- **Field[] getDeclaredFields()**
 - Renvoie **tous les attributs locaux uniquement**
- Ces deux méthodes renvoient un tableau de taille nulle si
 - Pas d'attributs
 - La classe est en fait un type prédéfini (int, double...)

Méthodes de Class (suite)

- **Method[] getMethods ()**
 - Ne renvoie que les méthodes public, locales et hérités
- **Method[] getDeclaredMethods ()**
 - Renvoie toutes les méthodes locales uniquement
- **Constructor[] getConstructors ()**
 - Ne renvoie que les constructeurs publics
- **Constructors[] getDeclaredConstructors ()**
 - Renvoie tous les constructeurs

Méthodes de Field, Method, Constructor

- `Class getDeclaringClass ()`
- `Class [] getExceptionTypes ()`
 - (Constructor et Method uniquement)
- `int getModifiers ()`
- `String getName ()`
- `Class [] getParameterTypes ()`
 - (Constructor et Method uniquement)

TP : un analyseur de classes

- Ecrire une méthode
analyseClasse (String nomClasse)
- qui affiche tout ce qu'il est possible d'obtenir
comme information sur la classe dont le nom
est passé en paramètre

Analyser un objet à l'exécution

Analyser un objet inconnu ?

- Jusqu'à présent, nous pouvons
 - déterminer les **noms** et les **types** des attributs d'un objet
 - Obtenir de l'objet, son type (un autre objet, de type **Class**)
 - Appeler **getDeclaredFields()** sur l'objet obtenu...
- Nous voulons maintenant obtenir le **contenu** d'un objet, c'est-à-dire le contenu de ses attributs
 - on ne connaît pas l'objet à examiner à l'avance, on ne connaît pas sa classe...

Accéder à la valeur d'un attribut

- Utiliser la méthode `get()` de la classe `Field`
 - Si `f` est un objet de type `Field`
 - `f = field[i];`
 - Si `obj` est un objet de la classe dont `f` est l'attribut
 - Alors `f.get(obj)` renvoie la valeur de l'attribut `f` de l'objet `obj`

Exemple

```
public class Personne {  
    private String nom;  
    private String prenom;  
    private int age  
    public Personne(String nom, String prenom, int age) {  
        this.nom = nom;  
        this.prenom = prenom;  
        this.age = age;  
    }  
}
```

Exemple (suite)

```
Personne p = new Personne("Chombier",  
    "Maurice", 47);
```

...

```
Class cl = p.getClass();
```

```
Field f = cl.getField("nom");
```

```
Object v = f.get(p);
```

```
System.out.println(v); // affiche "Chombier"
```

- Mais... Ca ne marche pas !
 - Car l'attribut "nom" est **private** !
 - Lève une **IllegalAccessException**

Exemple (suite)

- Le gestionnaire de sécurité Java permet de voir quels sont les attributs d'un objet, mais pas toujours de consulter leur valeur !
- Solution envisageable
 - mettre l'attribut "nom" public (bof)
 - mettre le code dans la classe Personne (bof bof)
 - utiliser **AccessibleObject.setAccessible(AccessibleObject[] array, boolean flag)**
 - Permet, par exemple, de donner accès aux attributs private
 - Peut aussi ne pas fonctionner si le gestionnaire de sécurité n'a pas donné l'autorisation (java.policy... cf. java rmi)
 - Lève alors une **SecurityException**

Exemple (suite)

- Que se passe-t-il si un attribut est d'un type prédéfini ?
 - `f.get(obj)` renvoie un **Object** !
- Dans ce cas, c'est la classe correspondante qui est choisie : **Integer**, **Double**, **Float**, etc...

```
Field f = cl.getField("age"); // age est un int
Object v = f.get(p); // v est un Integer
```
- Avec les JDK 1.3 et 1.4 :
 - `f.getInt()`, `f.getDouble()`, etc...
- Avec Java 5 :
 - L'Auto-boxing simplifie un peu la notation

TP : une méthode toString générique

- Souvent, on redéfinit toString pour décrire un objet en affichant la valeur de ses attributs
- Avec la réflexivité, il est possible
 - d'écrire une méthode une seule fois
 - de la placer dans le source des classes ou assez haut dans le graphe d'héritage
 - Le code n'est écrit qu'une seule fois !

Exemple d'utilisation : un tableau grossissant

La classe Array de java.lang.reflect

- Problème classique : un tableau d'objets
 - d'un certain type
 - Et le tableau est plein
- On veut le faire grossir !

```
Personne[] tab = new Personne[10];
```

```
...
```

```
// Le tableau est plein
```

```
tab = (Personne[]) grossitTableau(tab);
```

Comment s'y prendre ?

- Essayons ceci :

```
static Object[] grossitTableau(Object[] tab) {  
    int nouvelleTaille = tab.length * 11/10 + 10;  
    Object[] nouveauTableau = new Object[nouvelleTaille];  
    System.arraycopy(tab, 0, nouveauTableau, 0,  
        tab.length);  
    return nouveauTableau;  
}
```

- Problème : le type **Object []** ne peut *ici* être transtypé (*casté*) en **Personne []**
- Pourquoi ?

Méthodes de la classe Array et de la classe Class

- Dans Array
 - `static Object newInstance(Class componentType, int length)`
 - `static int getLength()`
 - `boolean isArray()`
- Dans Class
 - `Class getComponentType()` , ne s'applique que sur un `Array`
 - Ex : `Class type = o.getClass().getComponentType();`
 - `o` doit être un `Array`, et `type` représente le type des éléments du tableau.

TP : modifier le code précédent

- Solution : utiliser la réflexivité pour créer un nouveau tableau du même type que le tableau original
- A faire en TP : modifier la méthode **Object[] grossitTableau()**
- ... pour qu'elle fonctionne !
 - Penser à utiliser les méthodes de **Array** et de **Class** que nous venons de voir
 - Tester aussi avec un **int[]** ! Attention, piège !

Des pointeurs sur fonction ?

Passer un paramètre à une fonction, de type Method ?

- Qui a dit que Java n'avait pas de pointeurs sur fonction ?

```
public print(double debut, double fin, double pas,  
    Method f) {  
    for(double x = debut; x < fin; x+= pas) {  
        f(x);  
    }  
}
```

- Pas si simple !

Appeler une méthode

- La classe Method possède

```
Object invoke(Object o, Object[] args);
```

```
Object invoke(Object obj, Object... args); // Java 5
```

- `o` est l'objet dont qui va recevoir le message
 - Si on veut invoquer une méthode statique, `o` vaut `null`
- `args` est la liste des paramètres

- Exemple

- pour simuler `yoda.getNom()` où `yoda` est une instance de `Personne` et où `f` représente `getNom()`

```
String n = (String) f.invoke(yoda, null);
```


Invoquer une méthode (suite)

- Dans le cas de types prédéfinis, utiliser les "wrapper classes" **Integer**, **Float**, **Double**, etc...
- Exemple avec **f** qui représente **setAge(int age)** de la classe **Personne**

```
// JDK 1.2 à 1.4
```

```
Object[] args = {new Integer(735)};
```

```
f.invoke(yoda, args);
```

```
// Java 5 : Autoboxing + liste variable d'arguments
```

```
f.invoke(yoda, 735);
```

Comment obtenir un objet de type Method ?

- Utiliser `getMethod(...)` ou `getDeclaredMethods()` de la classe `Class`
- A cause de la surcharge, on doit préciser les types des paramètres avec **`getMethod(...)`**

```
Method getMethod(String nom, Class[] args)
```

```
Method getMethod(String nom, Class... args) // Java 5
```

- Exemple

```
m1 = Personne.class.getMethod("getNom", null);
```

```
m2 = Personne.class.getMethod("setAge",  
                               new Class[] {int.class});
```

```
// Java 5
```

```
m2 = Personne.class.getMethod("setAge", int.class);
```

Pointeur sur fonction ou pas ?

- Nous nous sommes bien amusés (sic) avec des pointeurs sur fonction
- Mais c'est bien sûr à éviter !
 - Déjà, en C/C++ c'est à éviter
 - Transtypage de partout (Object partout)
- nous ferons bien mieux avec l'héritage, les interfaces et les classes internes ! Bref, avec **le polymorphisme** !
- Et quand nous n'y arriverons plus ou difficilement, nous utiliserons des *Design Patterns*...

Autres fonctionnalités

- Classe Proxy
 - Des méthodes statiques pour créer dynamiquement des classes et des instances *proxy*
- Une classe proxy (dynamique) est une classe qui implémente une liste d'interfaces spécifiée à l'exécution
 - Sans qu'un **texte** de classe existe
- On peut ensuite créer des instances par réflexivité
- Bénéfices ?
 - Créer des instances compatibles avec plusieurs interfaces (pour s'adapter facilement), sans écrire de classes au préalable

Nouvelles fonctionnalités de réflexivité en Java 5

- Metadata...
 - Annotations @... Dans le code Java
 - Contenu des annotations accessibles par introspection à l'exécution
- Prise en compte de la généricité
 - Classe (notion de type et de type paramétré)
- Autoboxing des types de base
 - Facilite la récupération des valeurs de ces types

Annotations en Java

- Une annotation permet de *marquer* certains éléments du langage Java afin de leur ajouter une propriété particulière
- Ces annotations peuvent ensuite être utilisées à la compilation ou à l'exécution pour automatiser certaines tâches

@MonAnnotation

```
public class MaClasse {  
    /* ... */  
}
```

- Annotations *standards* : @Deprecated, @Override, @SuppressWarnings("deprecation")
- Méta-annotations (pour annoter d'autres annotations) :
 - @Documented, @Inherit
 - @Retention(RetentionPolicy.SOURCE ou .CLASS ou .RUNTIME)
 - ...

Création d'une annotation

- Création

```
import java.lang.annotation.*;  
import static java.lang.annotation.RetentionPolicy.SOURCE;
```

```
@Documented
```

```
@Retention(SOURCE)
```

```
public @interface TODO {  
    /* Message décrivant la tâche */  
    String value();  
}
```

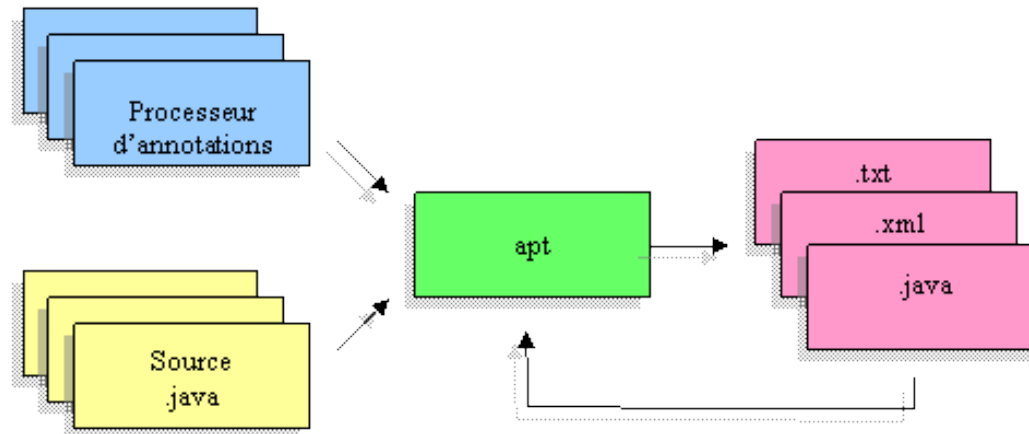
Attribut (type limité)

- Utilisation

```
@TODO(value="Coder tout ça vite fait...")  
public void ZeMethod () {  
    ...  
}
```

Exploitation des annotations

- apt (annotation processing tool), Java 5



© jmdoudoux

- Introspection : `getAnnotations` sur tous les éléments : Class, Method, etc.
- Compilateur Java (Java 6) : API Pluggable Annotation Processing
 - Plus besoin d'outil externe