Université Nice Sophia Antipolis

Réflexivité

F. Mallet

Adaptation du cours de Ph. Collet et M. Buffa

miage.m1@gmail.com

http://deptinfo.unice.fr/~fmallet/

Université Nice Sophia Antipolis

La classe Class

RTTI en java

Run-Time Type Identification

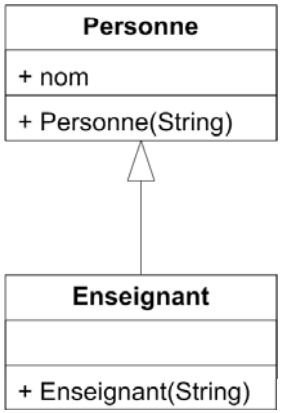
- Java maintient ce qu'on appelle l'Identification de Type au Run-Time (RTTI) sur tous les objets.
  - Permet de connaître le **type dynamique** d'une référence
    - Pas nécessairement le même que le **type statique**
  - Permet d'implémenter la liaison dynamique

Type statique **Type dynamique**

```
ObjetGraphique o = new Cercle();
o.draw();
```

- Deux phases
  - Compilation : type statique
  - Exécution : type dynamique (liaison dynamique)

La classe Class et le RTTI




```
Personne p1 = new Personne("Philippe R.");
Personne p2 = new Enseignant("Michel B.")

Class<? extends Personne> c1 = p1.getClass();
System.out.println(c1.getName() +" "+ p1.nom);

Class<? extends Personne> c2 = p2.getClass();
System.out.println(c2.getName() +" "+ p2.nom);
```

- Affiche

```
miage.m1.cm.Personne:Philippe R
miage.m1.cm.Enseignant:Michel B
```

Le champ statique **class**

- On peut aussi obtenir un objet de type Class :

```
Class<Enseignant> cl1 = Enseignant.class;
Class<Integer> cl2 = int.class;
Class<Double> cl3 = double.class;
…
```

- Utile pour tester le type (type prédéfini).
- Un objet de type **`Class`** décrit un TYPE, pas forcément une CLASSE !

Class<?> Class.forName(String)

- Charger une classe à partir de son nom

```
String nomClasse = "java.awt.image.BufferedImage";
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 = "Personne";
Class<?> cl = Class.forName(nomClasse);
Object o = cl.newInstance();
```

  - Appel du constructeur par défaut (sans paramètres)
  - Sinon **`newInstance(Object [] params)`** de la classe **`Constructor`** (**`java.lang.reflect`**)

Méthodes de la classe `Class<E>`

- `String getName()`
- `Class getSuperClass();`
- `Class[] getInterfaces();`
- `boolean isInterface();`
- `boolean isInstance(Object o);`
- `String toString();`
- `static Class forName(String name);`
- `E newInstance();`
- `boolean isAssignableFrom(Class<?> c);`
- `Class<? extends U> asSubClass(Class<U> c);`

Cf. Exemple de code

Université Nice Sophia Antipolis

Réflexivité et Introspection

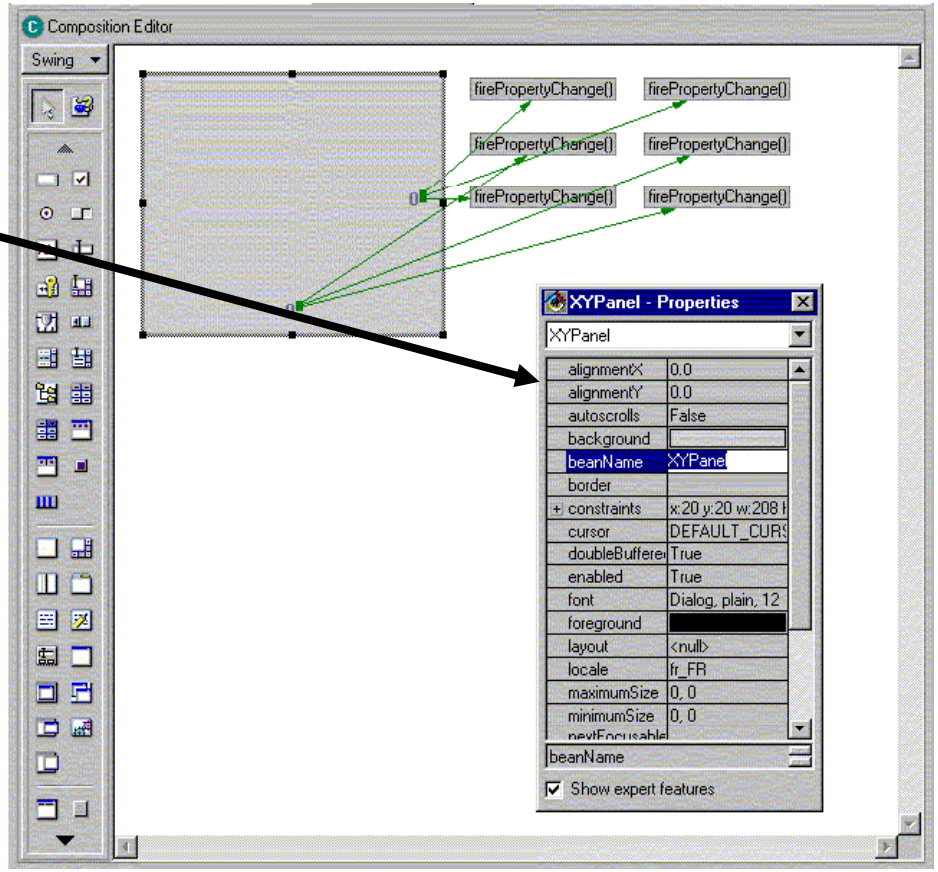
Université Nice Sophia Antipolis

Introduction/Définition

- Réflexivité
  - Capacité à se décrire soi-même
- Introspection
  - Mécanisme qui permet l'accès **dynamique** à la structure d'un programme (pas au comportement!)
- 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 (JBuilder, Visual Café, …) ont besoin d'interroger dynamiquement les nouvelles classes (*beans*)
  - Des classes voyagent dans les architectures distribuées et on doit les découvrir dynamiquement.

Introspection dans un outil RAD

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



Introduction/Définition (suite)

- Un programme qui peut analyser des classes est dit "réflexif".
- Le package qui fournit ces possibilités est **`java.lang.reflect`**
- Mécanismes très puissants. On peut par exemple:
  - **Analyser des classes dynamiquement,**
  - **Inspecter des objets dynamiquement,**
  - *Ecrire une méthode* **`toString()`** *générique,*
  - *Manipuler des tableaux génériques,*
  - *Manipuler des méthodes comme des pointeurs sur fonction du C/C++, etc…*

Réflexivité, Introspection et meta-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

Université Nice Sophia Antipolis

Analyser une classe

Université Nice Sophia Antipolis

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 et surtout des méthodes pour les exécuter

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 interpréter cette valeur.
    - **`String toString(int)`**
    - **`boolean isFinal(int)`**
    - **`boolean isPublic(int)`**
    - **`boolean isPrivate(int)`**
    - **`…`**

Université Nice Sophia Antipolis

Exemple

- Afficher la structure d'une classe (cf. TP)

```
class java.lang.Double extends java.lang.Number {
// Champs
   public static final double POSITIVE_INFINITY;
   public static final double NEGATIVE_INFINITY;
   public static final double NaN;
// Constructeurs
   public java.lang.Double(double);
   public java.lang.Double(java.lang.String);
// Méthodes
   public java.lang.String toString(double);
   public boolean isNaN(double);
   public boolean equals(java.lang.Object);
   …
}
```

Méthodes de la classe **`Class`**

- **`Field[] getFields()`**
  - Ne renvoie que les champs publics, locaux et hérités
- **`Field[] getDeclaredFields()`**
  - Renvoie tous les attributs locaux uniquement
- Ces deux méthodes renvoient un tableau de longueur nulle si
  - Pas de champs
  - La classe est un type prédéfini (int, double…)

Université Nice Sophia Antipolis

Méthodes de Class (suite)

- **`Method[] getMethods()`**
  - Ne renvoie que les méthodes publiques, locales et héritées
- **`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)

Université Nice Sophia Antipolis

Analyser un objet dynamiquement

Université Nice Sophia Antipolis

Analyser un objet inconnu

- Nous *avons vu* comment déterminer les **noms** et les **types** des champs d'un objet
  - Obtenir un objet de type **`Class`**
  - Appeler **`getDeclaredFields()`** sur l'objet obtenu
- Nous *allons voir* comment obtenir la **valeur** des champs d'un objet
  - Attention, on ne connaît pas l'objet à examiner à l'avance !!! (On ne connaît pas sa classe)

Accéder à la valeur d'un champ

- Utiliser la méthode **`get()`** de la classe **`Field`**
  - Si **`f`** est un objet de type **`Field`**
  - Si **`obj`** est un objet de la classe dont **`f`** est le champ…
  - Alors **`f.get(obj)`** renvoie la valeur de l'attribut **`f`** de l'objet **`obj`**

Université Nice Sophia Antipolis

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;
   }
}
```

Université Nice Sophia Antipolis

Exemple (suite)

- Un petit bout de code !

```
Personne p = new Personne("R.", "Philippe", 33);
…
Class cl = p.getClass();
Field f = cl.getField("prenom");
Object v = f.get(p);
System.out.println(v); // affiche "Philippe"
```

- Mais… Ca ne marche pas !
  - Le champ "prenom" est **`private`** !
  - Lance **`IllegalAccessException`**

Exemple (suite)

- On peut voir les champs d'un objet
  - Mais pas toujours consulter leur valeur
  - La sandbox interdit l'accès
- Solutions :
  - mettre l'attribut "prenom" public
  - mettre le code dans la classe **`Personne`**
  - Utiliser **`AccessibleObject.setAccessible(AccessibleObject[] array, boolean flag)`** (jdk1.2)
    ou méthode `setAccessible(boolean)` sur le champ
    - Donne accès aux champs privés

Exemple (suite)

- Un petit bout de code !

```
Personne p = new Personne("R.", "Philippe", 33);
…
Class cl = p.getClass();
Field f = cl.getField("prenom");
f.setAccessible(true);
Object v = f.get(p);
System.out.println(v); // affiche "Philippe"
```

- On peut modifier et lire des champs privés !
  - Seulement si c'est autorisé par le policy manager.

Exemple (suite)

- Avec les types prédéfinis ?
  - **`f.get(obj)`** renvoie un **`Object`** !
  - La valeur est encapsulée (*boxing*) dans : **`Integer`**, **`Double`**, **`Float`**, etc…

```
Field f = cl.getField("age"); // age est un int
Object v = f.get(p);          // v est un Integer
```

- Depuis JDK1.3: **`f.getInt()`**, **`f.getDouble()`**

- Depuis JDK1.1: **`Class.isPrimitive()`**

A faire : méthode toString générique

- Souvent, on redéfinit **`toString`** pour afficher la valeur de ses champs
- Avec l'introspection
  - on peut écrire une méthode toString générique une fois pour toute !
- A faire en TP !

Université Nice Sophia Antipolis

Exemple d'utilisation : un tableau grossissant !

La classe Array de java.lang.reflect

- Problème classique :
  - on a un tableau d'objets d'un certain type qui est plein,
- On veut l'agrandir !

```
String[] tab = {"philippe", "michel"};
…
// Le tableau est plein
tab = (String[]) grossitTableau(tab);
```

Comment s'y prendre ?

- Examinons ce bout de code

```
static Object[] grossitTableau(Object[] tab, int newLength) {
    Object[] nouveauTableau = new Object[newLength];
    System.arrayCopy(tab, 0, nouveauTableau,
                     Math.min(tab.length, newLength));
    return nouveauTableau;
}
```

- Problème :
  - le type **`Object[]`** ne peut pas être transtypé en **`String[]`** (cf. Démo)
  - Pourquoi ?

Méthodes de la classe Array et de la classe Class

- Dans Array (java.lang.reflect)
  - **`static Object newInstance(Class componentType, int length)`**
  - **`static int getLength()`**
- Dans Class
  - **`boolean isArray()`**
  - **`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.

Exercice à faire : modifier le code précédent

- Solution : utiliser l'introspection 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 !
  - JDK 1.6 introduit des méthodes **`copyOf`** pour répondre à ce problème récurrent de Java.

Université Nice Sophia Antipolis

Des pointeurs sur fonction ?

Passer une fonction en paramètre ?

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

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

- Pas si simple !!!

Invoquer une méthode

- La classe Method possède

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

  - **o** est l'objet dont on veut appeler la méthode
    - Si on veut invoquer une méthode statique, **o** vaut **null**
  - **args** est la liste des paramètres

- Exemple pour simuler **philippe.getNom()** où **philippe** est une instance de **Personne** et où **m** représente **getNom()**

```
String n = (String)m.invoke(philippe, null);
```

Invoquer une méthode (suite)

- Dans le cas (paramètre et résultat) de types primitifs
  - utiliser les classes enveloppantes **`Integer, Float, Double`**, …

- Exemple avec **`m`** qui représente **`setAge(int a)`** de la classe **`Personne`**

```
Object[] args = {new Integer(33)};
m.invoke(philippe, args);
```

Comment obtenir une 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)
```

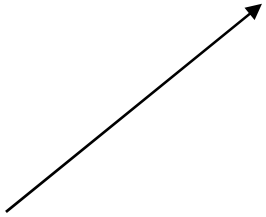
- Exemples

```
m1 = Personne.class.getMethod("getNom", null);
m2 = Personne.class.getMethod("setAge",
                              new Class[] {int.class})
```

Exercice

- Modifier la méthode **`print`** vue précédemment pour qu'elle fonctionne.
- Tester par exemple avec

```
print(1, 10, 1,
      Math.class.getMethod("sqrt",
      new Class[] {double.class})
);
```



La méthode **`Math.sqrt(double)`**

Pointeur sur fonction ou pas ?

- Bon, nous nous sommes bien amusés avec des pointeurs sur fonction
- Mais c'est bien sûr à éviter !
  - vérifications à l'exécution (au *run-time*)
- On peut faire bien mieux avec des objets
  - Les objets ont des champs mais aussi des méthodes
  - Passer un objet en paramètre, c'est donc passer une valeur mais aussi les méthodes qui vont avec

Université Nice Sophia Antipolis

Ressources

- Consultez les guides de Sun sur la réflexivité et les classes proxy (jdk1.3, non étudiées mais présentes dans java.lang.reflect)
  - http://java.sun.com/docs/books/tutorial/reflect/index.html
  - http://java.sun.com/j2se/1.3/docs/guide/reflection/proxy.html