Module de programmation avancée

Introspection, Annotations, Persistance

Master 1 MIAGE – Année 2025-2026

Philippe Lahire

Sommaire

- Réflexivité et Introspection
- Annotations
- Persistance / Sérialisation

- Compléments :
  - Rappel sur la gestion des flots de données
  - Chargement dynamique
  - Classes Internes

Lien avec les autres UE :
l'UE « *Cadrage d'un projet informatique* »

Agenda

- 4 séances de cours 2 heures
- 5 séances de TP/TD de 2 heures
- Une note de TP/TD
  - TP Noté
  - Réponses aux questions
  - Qualité des exercices
- Une note d’examen de contrôle continu

Utilisation de l'IA

- Pas de problème tant que vous comprenez et gardez un esprit critique sur le rendu de l'IA
- Sur la partie TP, pas de problème si vous voulez utiliser l'IA
- Mais :
  - Je dois vous évaluer de manière équitable
  - Je dois évaluer votre niveau de compréhension (pas celle de l'IA)

⇒ Une évaluation sur machine (sans restriction internet)
Une évaluation sur papier (compréhension + bouts de code)

Programmation Avancée

Réflexivité et Introspection

Philippe Lahire

Les langages réflexifs

Réflexion / Réflexivité

La réflexivité en programmation est la capacité d'un programme à se manipuler lui-même, à modifier son propre comportement ou à inspecter sa propre structure pendant l'exécution. Cela signifie que le programme peut réfléchir sur lui-même, examiner son propre code, ses variables, ses fonctions, ses classes, etc., et prendre des décisions ou effectuer des actions en conséquence.

Un langage réflexif peut utiliser :

- Introspection
- Intercession

Meta-programmation : La capacité d'un programme à créer, modifier ou manipuler d'autres programmes pendant l'exécution.

Exemple de mise en œuvre :

Un **protocole à métaobjets** est une technique en informatique qui consiste à faire de l'interprète d'un programme un objet de première classe, au même titre que ceux qui composent le programme. Il est ainsi possible de le réécrire afin de changer l'interprétation du programme.

Rectangle a;… a.resize(2,3)

Introspection vs intercession

Introspection

L'introspection est la capacité d'un programme à examiner et à analyser sa propre structure et ses propres composants pendant l'exécution. Cela signifie que le programme peut inspecter ses variables, ses fonctions, ses classes, etc., et en extraire des informations sur leur type, leur contenu, leurs attributs, etc. L'introspection est très utile pour la débogage, la vérification de types, la génération de documentation automatique, et pour créer des outils de développement plus avancés.

En Python, l'introspection est facilitée par diverses fonctions et méthodes intégrées, telles que `dir()`, `type()`, `isinstance()`, `getattr()`, `hasattr()`, etc. Ces fonctions permettent aux développeurs d'inspecter les objets, les classes, les modules, et d'en extraire des informations détaillées.

Intercession

L'intercession, quant à elle, est la capacité d'un programme à intervenir dans les opérations effectuées sur ses composants, tels que les accès aux attributs, les appels de méthodes, etc. L'intercession permet aux développeurs de modifier ou d'étendre le comportement des objets ou des classes à la volée, sans avoir à modifier leur définition originale. Cela peut être très utile pour l'implémentation de mécanismes de sécurité, de journalisation, de caching, etc.

Aspect-Oriented Programming

En Python, l'intercession est principalement réalisée à l'aide de méthodes spéciales dans les classes, telles que `__getattribute__`, `__setattr__`, `__delattr__`, `__getattr__`, etc. Ces méthodes sont appelées automatiquement par Python lorsque certaines opérations sont effectuées sur les instances de la classe, permettant ainsi à la classe d'intervenir dans ces opérations.

Un exemple de AspectJ (AOP)

```java
import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.annotation.After;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
import org.aspectj.lang.annotation.Pointcut;

@Aspect
public class LoggingAspect {

    @Pointcut("execution(* *(..))")
    public void logMethod() {}

    @Before("logMethod()")
    public void logBefore(JoinPoint joinPoint) {
        System.out.println("Entrée dans la méthode " + joinPoint.getSignature().getName());
    }

    @After("logMethod()")
    public void logAfter(JoinPoint joinPoint) {
        System.out.println("Sortie de la méthode " + joinPoint.getSignature().getName());
    }
}
```

`@Aspect` : annotation qui indique que la classe `ExempleAspect` est un aspect.

`@Pointcut` : annotation qui définit un point de coupe pour l'aspect. Dans cet exemple, le point de coupe est défini pour intercepter l'appel de toutes les méthodes (`execution(* *(..))`).

`@Before` : annotation qui indique que le conseil (le code de l'aspect) doit être exécuté avant l'appel de la méthode.

Classification de quelques langages réflexifs

- Python : Introspection et Intercession
- Javascript : introspection et une certaine forme d'intercession (proxy)
- Ruby: Introspection et Intercession (*alias_method, define_method, méthodes magiques*)

  Conversions, initialisation, comparaison, etc.
- PHP : Introspection et Intercession (*méthodes magiques*)

  Accès, mise à jour d'une propriété, destruction objet
- C# : introspection et une certaine forme d'intercession

  Castle
- Java: introspection (persistance: une certaine forme d'intercession)

Mais aussi Perl, Groovy, R ….

Classification de quelques langages réflexifs

catégorie	Python	JavaScript	Ruby	PHP	C#	Java
Infos attribut	x	x	x	x	x	étendues
Lire valeur attribut	x	x	x	x	x	x
Modif. Valeur attribut	x	x	x	x	x	x
infos type	Nom uni.	Nom uni.	x	x	x	x
infos classe	basique		x	x	x	x
infos méthode	source		x	x	x	x
infos paramètres	x		x	x	x	x
Exécution méthode	x	x	x	x	x	x
infos parents/sousclasses	x		x	x	x	x
infos classes internes				Traits (framework)	oui	x
Ajout dans méthode	x	N (proxy)	*Méth magiques*	*Méth magiques*	*Castle*	non
Ajout/supp méthode	x	x	*define_method*	non	non	non
Ajout/supp attribut	x	x		non	non	non
Impl. constructeur		x		x	x	x

Python et l’introspection

Utilisation de `getattr` et `setattr`

Les fonctions `getattr` et `setattr` permettent de récupérer et de définir des attributs d'un objet de manière dynamique.

python Run Enregistrer Copie

```python
class MyClass:
    def __init__(self, a, b):
        self.a = a
        self.b = b

obj = MyClass(1, 2)
print(getattr(obj, 'a'))  # Output: 1

setattr(obj, 'a', 10)
print(obj.a)  # Output: 10
```

Python et l’intercession

```python
class Personne:
    def __init__(self, nom, age):
        self.nom = nom
        self.age = age

    def __getattribute__(self, name):
        if name == "nom":
            print("Vous essayez d'accéder au nom")
        elif name == "age":
            print("Vous essayez d'accéder à l'âge")
        return super().__getattribute__(name)

personne = Personne("John", 30)
print(personne.nom)  # Affiche "Vous essayez d'accéder au nom" puis "John"
print(personne.age)  # Affiche "Vous essayez d'accéder à l'âge" puis 30
```

JavaScript et l’introspection

javascript Enregistrer Copie

```javascript
class MyClass {
  constructor(a, b) {
    this.a = a;
    this.b = b;
  }

  myMethod(x, y) {
    return x + y;
  }
}

const obj = new MyClass(1, 2);

console.log(typeof obj); // "object"
console.log(obj instanceof MyClass); // true

console.log(Object.keys(obj)); // ["a", "b"]
console.log(Object.values(obj)); // [1, 2]
console.log(Object.entries(obj)); // [["a", 1], ["b", 2]]
```

Ruby et l’introspection

Obtenir la classe d'un objet

ruby Enregistrer Copie

```ruby
obj = "Hello, World!"
puts obj.class             # => String
```

Obtenir les méthodes d'un objet

ruby Enregistrer Copie

```ruby
puts obj.methods.sort      # => ["!", "===", "==", "...", "class", "clone", "display", "dump", "eql?",
```

Obtenir les méthodes propres d'un objet (c'est-à-dire, les méthodes définies dans la classe de l'objet et non héritées)

ruby Enregistrer Copie

```ruby
puts obj.public_methods(false).sort  # => ["+", "[]", "capitalize", "casecmp", "center", "chomp", "cho
```

PHP et l'introspection

```php
class MyClass {
    public $publicProperty = 'public';
    private $privateProperty = 'private';

    public function publicMethod() {
        return 'public method';
    }

    private function privateMethod() {
        return 'private method';
    }
}
```

```php
$reflectionClass = new ReflectionClass('MyClass');

// Obtenir les propriétés
$properties = $reflectionClass->getProperties();
foreach ($properties as $property) {
    echo $property->getName() . "\n";
}

// Obtenir les méthodes
$methods = $reflectionClass->getMethods();
foreach ($methods as $method) {
    echo $method->getName() . "\n";
```

Invocation Dynamique de Méthodes

php Enregistrer Copie

```php
$reflectionMethod = new ReflectionMethod('MyClass', 'publicMethod');
$instance = new MyClass();
echo $reflectionMethod->invoke($instance); // Appelle la méthode publicMethod sur l'instance de MyClas
```

C# et l’introspection

csharp Enregistrer Copie

```csharp
using System;
using System.Reflection;

public class Person
{
    public string Name;
    public int Age;
}

public class Program
{
    public static void Main()
    {
        Type type = typeof(Person);
        FieldInfo[] fields = type.GetFields();
        foreach (FieldInfo field in fields)
        {
            Console.WriteLine(field.Name);
        }
    }
}
```

Obtenir le Type de la Classe :

```csharp
Type type = myObject.GetType();
```

Cette ligne obtient le type de l'instance `myObject` .

Obtenir l'Information sur la Méthode :

```csharp
MethodInfo sayHelloMethod = type.GetMethod("SayHello");
```

Cette ligne obtient l'information sur la méthode `SayHello` de la classe `MyClass` .

Invoker la Méthode avec des Arguments :

```csharp
sayHelloMethod.Invoke(myObject, new object[] { "Alice" });
```

Et dans Java…

Java = introspection réflexive, pas de protocole de méta-programmation

Paquetage *java.lang.reflect (*et *java.lang)*
+
Des éléments syntaxiques

Ce n’est pas ici de l’introspection

```java
public class Personne {

        private String nom;
        private int age;

        public Personne() {
            System.out.println("Constructeur par défaut de personne");
        }

        public Personne (String n, int a) {
            nom = n;
            age = a;
        }

        public String getNom() {
            return nom;
        }
        public void setNom(String nom) {
            this.nom = nom;
        }
}
```

```
Class TestE  {
public static void main(String[] args) {
  Personne p = new Personne(« Alice »,30);
  System.out.println(p.nom);
  p.nom = « bob »;
}
```

C ’est de l’introspection

```java
public class Personne {

        private String nom;
        private String prenom;
        private int age;

        public Personne() {
            System.out.println("Constructeur par défaut de personne");
        }

        public Personne (String n, String p, int a) {
            nom = n;
            prenom = p;
            age = a;
        }

        public String getNom() {
            return nom;
        }
        public void setNom(String nom) {
            this.nom = nom;
        }
}

```

```java
Personne personneInstance = new Personne("Alice", 30);

Class<?> personneClass = personne.getClass();

Field nomField = personneClass.getDeclaredField("nom");
nomField.setAccessible(true);
System.out.println("Nom: " + nomField.get(personneInstance));

nomField.set(personneInstance, "Bob");
```

Réflexivité et introspection : Quelle utilisation ?

- Affichage d'informations sur le contenu des classes
- Exécution de méthodes
- Modification/consultation de la valeur des champs
- …

Pas de nom de classe dans le code…

- Débogage de programmes
- JUNIT
- Utilisation couplée avec les annotations :
  - JPA (Java Persistence API): Mise en correspondance modèle Objet/Relationnel
    - Hibernate, OpenJDO
  - Framework Spring
  - ….

<u>Au niveau macro</u> :
Environnements de développement
Echange de classe dans les systèmes distribués
Programmation d'outils spécifiques

JUNIT : Un exemple d'utilisation de l'introspection (1)

- JUnit : framework open source (https://junit.org/)
- développement et l'exécution de tests unitaires automatisables.
- s'assurer que le code répond toujours aux besoins même après d'éventuelles modifications (tests de non-régression)
- Fil conducteur :
  - séparer le code de la classe, du code qui permet de la tester.

    Pas de méthode main dans chaque classe pour embarquer les tests
  - Les exprimer dans des classes dédiées
  - sous la forme de cas de tests avec leurs résultats attendus.
  - Faire exécuter les différents tests à JUNIT et déclencher une exception en cas d'erreur
- Utilisation de l'introspection pour la mise en œuvre

JUNIT3 : Un exemple d'utilisation de l'introspection (2)

```
public class ExempleClasse {
  public static int addSiPetit (int a, int b) {
      ! Le résultat doit être inférieur à 100
      int res = a + b;
      if (res >100) res = -1;
      return res;
    }
}
```

```
import junit.framework.*;
public class ExempleClasseTest extends TestCase {
  public void testAddSiPetit1() throws Exception {
    assertEquals(2, ExempleClasse. addSiPetit(1,1)); }
  public void testAddSiPetit2() throws Exception {
    assertEquals(-1, ExempleClasse. addSiPetit(90,11));
  }
```

```
java -cp junit.jar;. junit.textui.TestRunner ExempleClasseTest
```

Recherche par introspection :

- Classe de test (nom = ….TEST
- Méthode de test :
  - Nom = test…
  - + règles suivantes : public, pas de retour, pas de paramètre

JUNIT4 : Un exemple d'utilisation de l'introspection (2)

```
public class ExempleClasse {
  public static int addSiPetit (int a, int b) {
      ! Le résultat doit être inférieur à 100
    int res = a + b;
    if (res >100) res = -1;
    return res;
    }
}
```

JUNIT5 : annotations différentes + nombreuses… @BeforeAll….

```
import junit.framework.*;
public class ExempleClasseTest extends TestCase {
@Test
  public void addSiPetit1() throws Exception {
    assertEquals(2, ExempleClasse. addSiPetit(1,1));
  } }
```

```
java -cp junit.jar;. junit.textui.TestRunner ExempleClasseTest
```

Recherche par introspection :

- Classe de test (nom = ….TEST
- Méthode de test :
  - Une annotation « Test » avant la méthode
  - + règles suivantes : public, pas de retour, pas de paramètre

Mais aussi annotations :

- *@Before* ou *@After*
- *@BeforeClass* ou *@AfterClass*

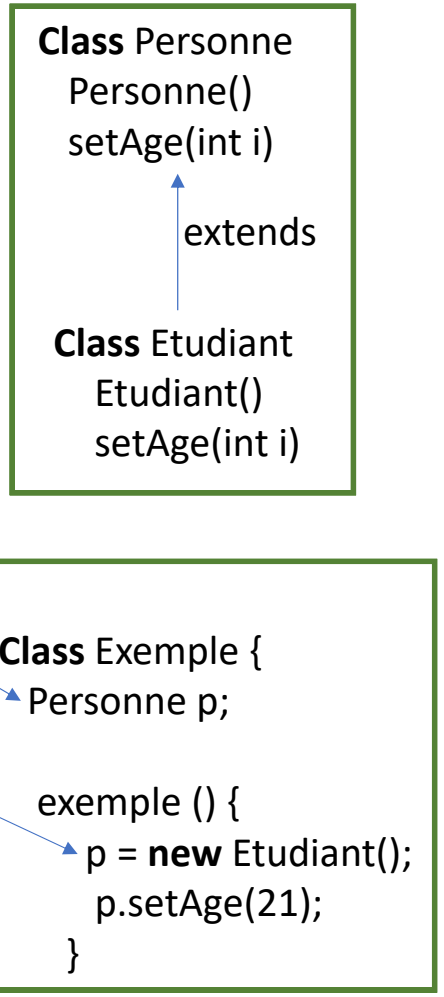
Types statiques et types dynamiques

Types d'une référence ou d'un attribut/champ

- Type statique de « p » : Personne
- Type dynamique de « p » :
  Liaison dynamique : Etudiant

De la compilation à l'exécution

- Type statique :
  - Vérification à la compilation:
    setAge existe bien dans Personne
- Type dynamique de « p » :
  - Liaison dynamique réalisée à l'exécution :
    setAge: Version de la classe Etudiant



Run-Time Type Identification (RTTI)

- Java maintient ce qu'on appelle l'Identification de Type à l'exécution sur tous les objets
- Permet de connaître le **type dynamique** d'une référence
- La classe Class et le RTTI

Permet l'accès à « getName() »

Permet de faire du contrôle de type

```
Personne p1 = new Personne("Philippe L.");
Personne p2 = new Etudiant(" Henri L.")
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 à l'exécution

```
miage.m1.cm.Personne:Philippe L.
miage.m1.cm.Etudiant:Henri L.
```

getClass retourne la classe obtenue par liaison dynamique

Champ statique « class »

- C’est une construction du langage
- Tout se passe comme si toute classe ou type primitif ou tableau avait un champ statique « class »
- Un objet de type class: un type, pas forcément une classe
- Exemple de compatibilités de type :

```
Class<Etudiant> cl1 = Etudiant.class;
Class<Integer> cl2 = int.class;
Class<Double> cl3 = double.class;
Class<Void> cl4 = void.class
Class<Etudiant[]> cl5 = Etudiant[].class
```

nomClasse.class
nomClasse[].class
typePrimitif.class
void.class

Comment décoder *getName()* de *Class*

java Enregistrer Copie

```java
public class Example {
    public static void main(String[] args) {
        System.out.println(int[].class.getName()); // "[I"
        System.out.println(byte[].class.getName()); // "[B"
        System.out.println(short[].class.getName()); // "[S"
        System.out.println(long[].class.getName()); // "[J"
        System.out.println(float[].class.getName()); // "[F"
        System.out.println(double[].class.getName()); // "[D"
        System.out.println(boolean[].class.getName()); // "[Z"
        System.out.println(char[].class.getName()); // "[C"
        System.out.println(int[][].class.getName()); // "[[I"
        System.out.println(int[][][].class.getName()); // "[[[I"
    }
}
```

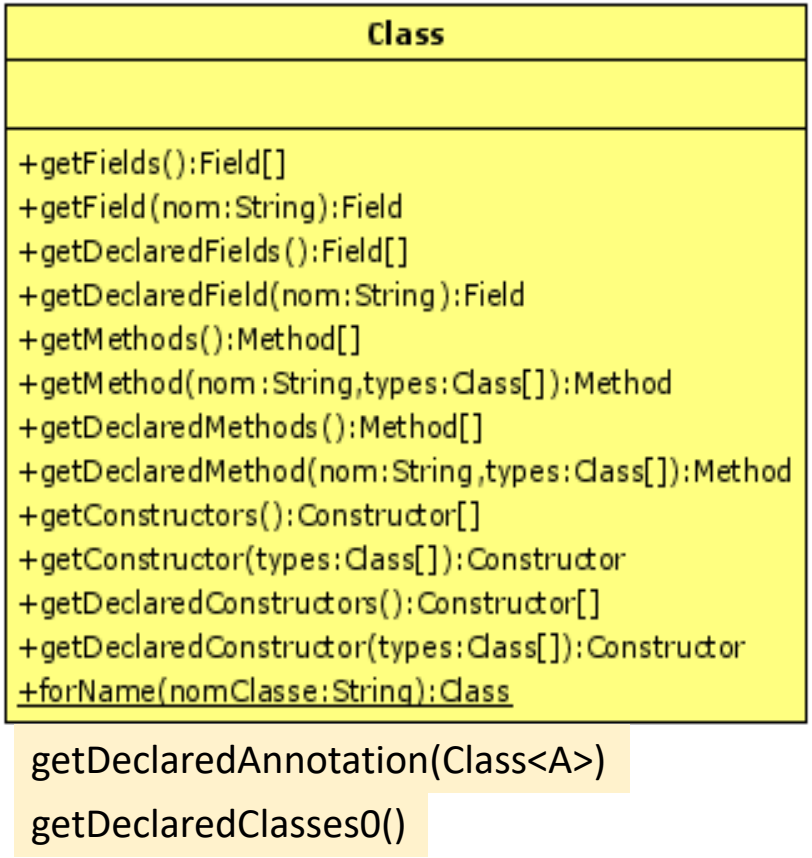
B : byte
S : short
I : int
J : long
F : float
D : double
Z : boolean
C : char

- `String st = "mon test";`
- `System.out.println("string: "+String.class.getName());`
- `System.out.println("string2: "+st.getClass().getName());`
- `System.out.println("string[]: "+String[].class.getName());`

```
string: java.lang.String
string2: java.lang.String
string[]: [Ljava.lang.String
```

La classe Class (hiérarchie et quelques fonctionnalités)

```
java.lang.Object
   |
   |-- java.lang.reflect.AnnotatedElement
   |     |
   |     |-- java.lang.reflect.AccessibleObject
   |     |     |
   |     |     |-- java.lang.reflect.Constructor
   |     |     |-- java.lang.reflect.Field
   |     |     |-- java.lang.reflect.Method
   |     |
   |     |-- java.lang.reflect.Class
   |     |-- java.lang.reflect.GenericDeclaration
   |           |
   |           |-- java.lang.reflect.Class
   |           |-- java.lang.reflect.Method
   |
   |-- java.lang.reflect.Type
   |     |
   |     |-- java.lang.reflect.Class
   |     |-- java.lang.reflect.ParameterizedType
   |     |-- java.lang.reflect.TypeVariable
   |     |-- java.lang.reflect.WildcardType
   |
   |-- java.lang.reflect.Member
   |     |
   |     |-- java.lang.reflect.Constructor
   |     |-- java.lang.reflect.Field
   |     |-- java.lang.reflect.Method
```



getDeclaredAnnotation(Class<A>)

getDeclaredClasses0()

Autres méthodes de la classe « Class <T>»

- `String getName(): nom de la classe`
- `String toString();`
- `public native Class<? super T> getSuperclass()`
- `public Class<?>[] getInterfaces()`
- `public native boolean isInterface();`
- `public native boolean isInstance(Object obj); (= instanceof)`
- `public native boolean isAssignableFrom(Class<?> cls)`
- `public <U> Class<? extends U> asSubclass(Class<U> clazz)`
- `public Class<?> getComponentType()`

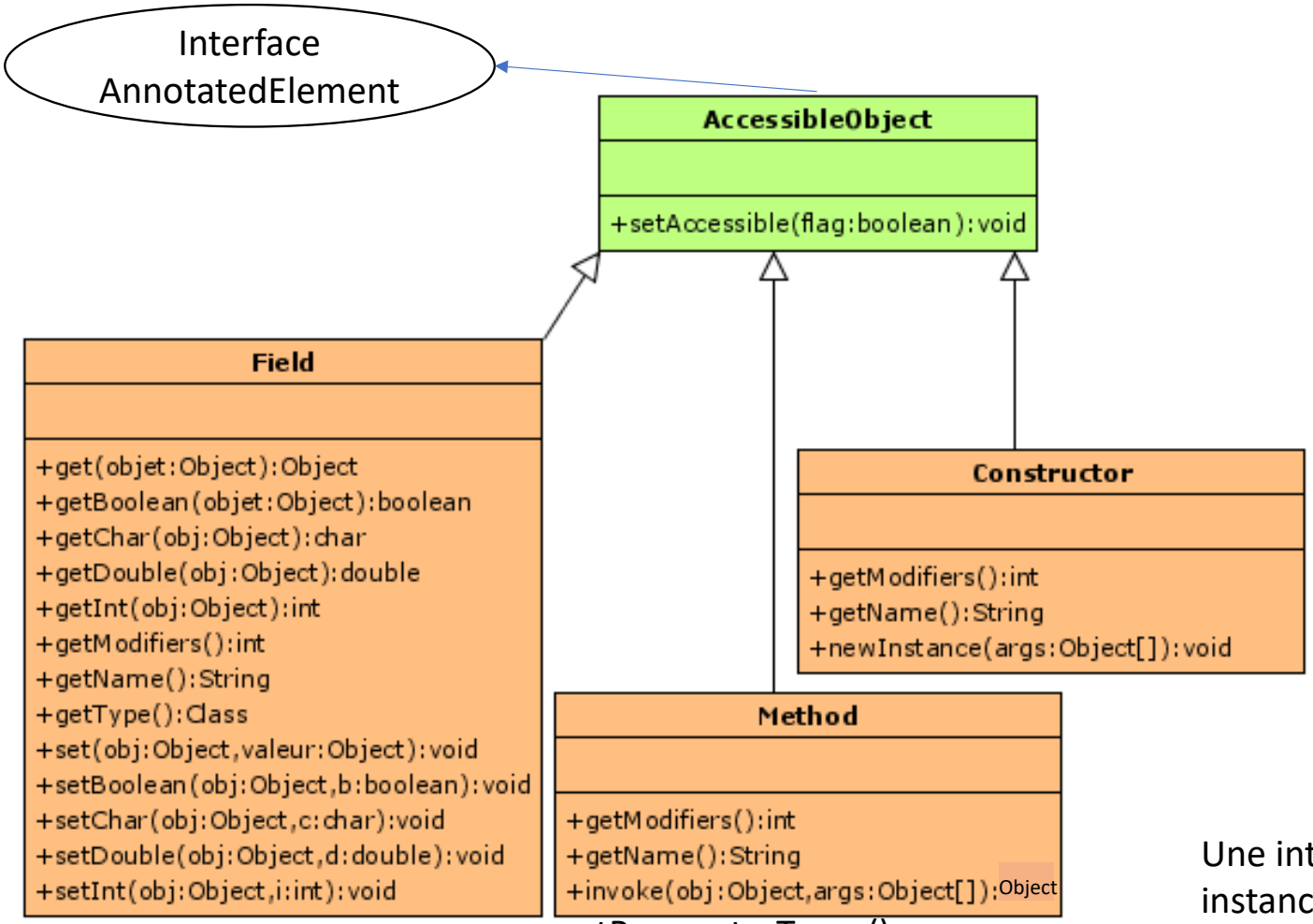
  si le type de la classe est un tableau

Method getEnclosingMethod()
getEnclosingclass()

Une interface est représentée par une instance de la classe Class

```
getName()
getCanonicalName()
getSimpleName()
getPackage()
getSuperclass()
getInterfaces()
getConstructors()
getMethods()
getFields()
getDeclaredConstructors()
getDeclaredMethods()
getDeclaredFields()
newInstance()
isInstance(Object obj)
isAssignableFrom(Class<?> clazz)
isPrimitive()
isArray()
isEnum()
isInterface()
isMemberClass()
```

Description des éléments d'une classe



getParameterTypes()

Accès à toutes les informations :
- champs,
- méthodes,
- Annotations
- Paramètres des méthodes
- Classes imbriquées
- Etc.

<u>Autres méthodes de *Method*</u> :
Class<?>[] getParameterTypes()
int getParameterCount()
Class<?> getReturnType
Type getGenericReturnType()
Method getRoot()
boolean isDefault() dans les interfaces, non abstraites, liens avec l'héritage multiple

Une interface est représentée par une instance de la classe Class

Quelques Rappels de la séance précédente

- De nombreux langages propose de la réflexivité
  - Introspection: voir les structures vs voir les objets
  - Intercession : ajouter du comportement à une méthode / à l'appel d'une méthode
- Utilisation: Débogage (IDE), tests unitaires (JUNIT), préoccupations transversales
- Java :
  - Classe Class: rappel contenu….
  - Classes Field (get, set), Method (invoke), Constructor (newInstance)
  - Limitations: corps des classes

Charger une classe à partir de son nom

public static Class<**?**> forName(**String** className)

nomClasse : peut être un nom d'interface ou de classe

```
String nomClasse = "java.lang.reflect.Array";
try {
Class<?> cl = Class.forName(nomClasse);
Method[] md = cl.getDeclaredMethods();
System.out.println(md[0].getName());
} catch (ClassNotFoundException x) {}
```

➡ Utile pour charger des classes dont on ne connaît pas le nom à l'avance

Créer des instances sans utiliser « new »

```
public T newInstance(Object ... initargs)
   throws InstantiationException, IllegalAccessException,
          IllegalArgumentException, InvocationTargetException
```

nomClasse : peut être un nom d'interface ou de classe

```
Object o = cl.newInstance();
(deprecated)
```

```
String nomClasse = « miage.m1.cm.Personne";
try {
Class<?> cl = Class.forName(nomClasse);
Constructor<?>[] cc = cl.getConstructors();
Object o = cc[0].newInstance(…);
} catch (ClassNotFoundException | … x) {}
```

Objet cible + Dépend du constructeur si plusieurs dans la classe

⟹ Utile pour créer des instances de classes dynamiquement

Voir le contenu d'un champ « public »

```
String nomClasse = "miage.m1.td1.Personne";
try {
      Class<?> cl =    Class.forName(nomClasse);
      Constructor<?>[] cc = cl.getConstructors();
      Object o = cc[0].newInstance();
      Field f = o.getClass().getField("age");
      System.out.println(f.getInt(o));
} catch (ClassNotFoundException | … y) {}
```

- Un objet de type Field ne contient pas la valeur de l'attribut mais la « structure » de l'attribut
- *f.getType()* pour récupérer le type statique
- *f.get(o).getClass.getTypeName()* pour le type dynamique

Modifier le contenu d'un champ « public »

```
String nomClasse = "miage.m1.td1.Personne";
try {
      Class<?> cl =    Class.forName(nomClasse);
      Constructor<?>[] cc = cl.getConstructors();
      Object o = cc[0].newInstance();
      Field f = o.getClass().getField("age");
      f.set(o,2);
      System.out.println("valeur:" + f.get(e));
} catch (ClassNotFoundException | … y) {}
```

- Un objet de type Field ne stocke pas la valeur de l'attribut mais la « structure » de l'attribut
- f.getType pour récupérer le type statique
- f.get(o).getClass.getTypeName() pour le type dynamique

Si besoin

Cas d'une classe avec des types prédéfinis

- Types prédéfinis : char, boolean, float, long, int, short, byte
  Des valeurs, pas des objets
- Des classes associées: Float, Double, Integer, Short, Long, Character, Byte
  Encapsulation de la valeur (boxing) et opération inverse (unboxing)
- Accès aux valeurs
  - `f.get(obj)` renvoie un **`Object`** !
  - La valeur lue est encapsulée
  - Exemple :
  - **`Field f = cl.getField("age"); // age est un int`**
  - **`Object v = f.get(p);  // v est un Integer`**
- La classe Field contient des primitives dédiées :

getBoolean (Object): boolean	getInt (Object): int
getChar (Object) : char	getFloat (Object) : float

…

Cas d'une classe paramétrée (générique)

Exemple :

```
Exemple e;
e = new exemple();
Field f = e.getClass().getDeclaredField("tab_p");
Type t = f.getGenericType();
System.out.print("type:");
System.out.println(t.getTypeName());
```

```
Class Exemple {
  public ArrayList<Personne> tab_p;

   exemple () {
       tab_p = new ArrayList();
    }
```

Affichage :

```
type:java.util.ArrayList<miage.m1.td1.Personne>
```

Invoquer dynamiquement une méthode (1)

Utiliser de java.lang.reflect.Method

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

Une solution pour simuler : p.getNom() où ‘p’ une instance de Personne et ‘m’ représente la méthode getNom()

```
Method m = Personne.class.getMethod("getNom", null);
String n = (String) m.invoke(p, null);
```

Invoquer dynamiquement une méthode (2)

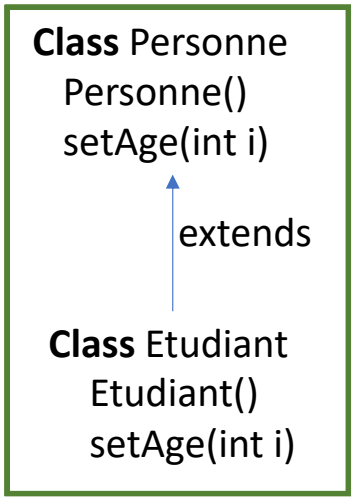
- Dans le cas (paramètre et résultat) de types primitifs
  utiliser les classes enveloppantes **`Integer, Float, Double`**, …

- Si **`m`** représente **`setAge(int a)`** de la classe **`Personne`**

```
Method m = Personne.class.getMethod("setAge", Integer.class);
Object[] args = {new Integer(33)};
m.invoke(p, args);
```

Deux points de vue

- Analyse du contenu d'une classe



- Analyse du contenu d'un objet

```
Personne p1 = new Personne("Philippe L.");
Personne p2 = new Etudiant(" Henri L.")
```

Sujet du premier TD

TD1 - Exercice 1 : Analyse du contenu d'une classe

```
public final class java.lang.Class<T> implements java.io.Serializable, java.lang.reflect.GenericDeclaration, java.lang.reflect.Type, java.lang.reflect.AnnotatedElement {
  transient java.lang.ClassValue$ClassValueMap classValueMap;
  public java.lang.String toString();
  public java.lang.String toGenericString();
  public static java.lang.Class<?> forName(java.lang.String) throws java.lang.ClassNotFoundException;
  public static java.lang.Class<?> forName(java.lang.String, boolean, java.lang.ClassLoader) throws java.lang.ClassNotFoundException;
  public static java.lang.Class<?> forName(java.lang.Module, java.lang.String);
  public T newInstance() throws java.lang.InstantiationException, java.lang.IllegalAccessException;
  public native boolean isInstance(java.lang.Object);
  public native boolean isAssignableFrom(java.lang.Class<?>);
  public native boolean isInterface();
  public native boolean isArray();
  public native boolean isPrimitive();
  public boolean isAnnotation();
  public boolean isSynthetic();
  public java.lang.String getName();
  public java.lang.ClassLoader getClassLoader();
```

.....

TD1 - Exercice 1 : Analyse du contenu d'une classe

```
public class AnalyseurDeClasse {

  public static void analyseClasse(String nomClasse) throws ClassNotFoundException {
    Class cl = …

    afficheEnTeteClasse(cl);

    afficheInnerClasses(cl);

    afficheAttributs(cl);

   afficheConstructeurs(cl);

   afficheMethodes(cl);

    // L'accolade de fin de classe !
    System.out.println("}");
  }
….
```

Analyser une classe (1): la structure d'une classe

Deux classes dans Java.Lang:

- La classe Class
- La classe Object

Dans java.lang.reflect**:**

- Trois classes : Field**,** Method**,** Constructor
- Mais aussi d'autres: Modifier, etc.

- Les quatre classes (*Class*…*Constructor*) 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

➡ Object et Class sont le point de départ qui permet d'accéder à « tout »

Analyser une classe (2) : reconnaitre les « modifiers »

- Field, Method, Constructor possèdent getModifiers() qui renvoie un int,
  Interprétation de l'entier bit par bit (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)
  - …
- Field, Method, Constructor (d'autres méthodes):
  - Class getDeclaringClass()
  - Class[] getExceptionTypes() et Class[] getParameterTypes()
    Constructor et Method uniquement (***Executable***)

Analyser une classe (3): Les champs

- 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…)

Quelques informations complémentaires :

- Le lieu de la déclaration (classe): *getDeclaringClass()*
- Le type « statique » du champ (Classe, primitif ou générique): *getType()* et *getGenericType()*
- Annotations (*voir chapitre du cours concerné*)

Analyser une classe (4): les méthodes / constructeurs

- 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
- Constructor[] getDeclaredConstructors()
  Renvoie tous les constructeurs
- Accéder aux paramètres : classe Parameter
  - Assez proche des fonctionnalités offertes par la clase FIELD: *getType*, *getParameterizedType*, annotations, etc.

TD1 - Exercice 2 : Réaliser une méthode toString() générique

Utiliser l'introspection pour créer une méthode toString() générique. Il s'agit en pratique de faire une méthode toString qui prend en paramètre un objet de type Object et affiche la valeur de chacun de ses champs. Attention, si les champs sont des références sur d'autres objets on descendra en profondeur pour afficher "récursivement" leur valeur également.

Pour permettre « l'arrêt » du programme, la méthode toString aura un deuxième paramètre qui est la profondeur à laquelle on souhaite descendre.

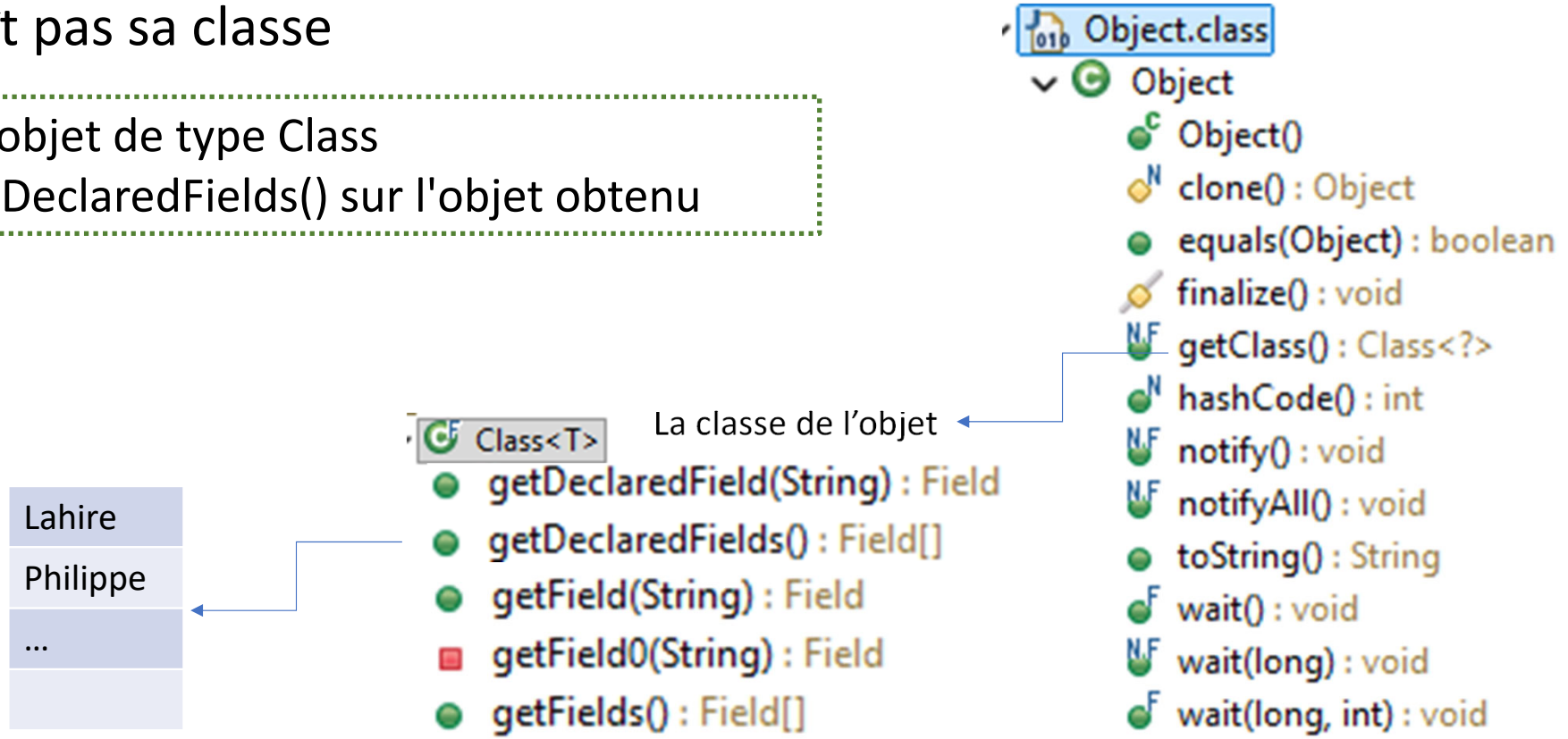
.....

Analyser un objet de type inconnu (1)

Comment obtenir la valeur des champs d'un objet

- on ne connaît pas l'objet à examiner à l'avance
- On ne connaît pas sa classe

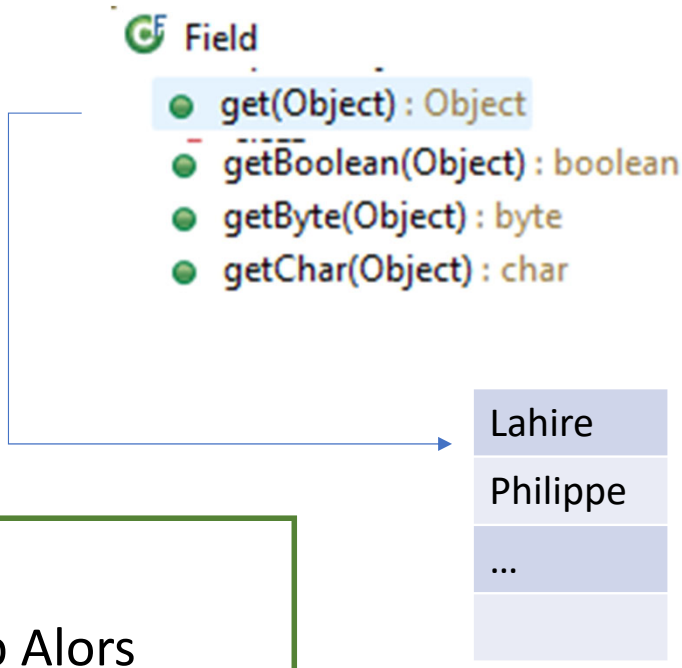
Obtenir un objet de type Class
Appeler getDeclaredFields() sur l'objet obtenu



Analyser un objet de type inconnu (2)

Accéder à la valeur d'un champ

- Utiliser la méthode get(Object obj) de la classe Field
  - Un type primitif la valeur est encapsulée dans un objet
  - Un champ statique: le paramètre est ignoré
  - Le champ doit être accessible (dépend des « modifiers »)
  - Gestion d'exceptions: accès illégal, type non conforme



Si **f** est un objet de type **Field** Alors
    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**

Analyser une instance de type connu (1)

On connait une instance et le nom des champs :

```
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);
```

On ne peut pas accéder à tout

Par défaut, le mécanisme de réflexion respecte le contrôle des accès.

- Exemple :Le champ "prenom" est private !
- Génération d'une exception: *IllegalAccessException*

```
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;
    }
}
```

Analyser une instance de type connu (2)

On peut voir les champs d'un objet mais pas toujours consulter leur valeur

Solutions de contournement :

- mettre l'attribut "*prenom*" public
- mettre le code dans la classe Personne
- si un programme Java n'est pas contrôlé par un gestionnaire de sécurité (*policy manager*) qui le lui interdit, il peut outrepasser son droit d'accès.

  Invoquer la méthode setAccessible(boolean) d'un objet Field, Method ou Constructor
- Autre méthode (méthode statique) :

  AccessibleObject.setAccessible(AccessibleObject[] array, boolean flag)

Analyser une instance de type connu (3)

Code mis à jour :

```
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);
```

Utilité des mécanismes réflexifs (1)

Exemple d'utilisation : Augmenter la taille d'un tableau

```
static Object[] AgrandirTailleTableau(Object[] tab, int newLength) {
          Object[] newTab = new Object[newLength];
          System.arraycopy(tab, 0, newTab, 0,
                           Math.min(tab.length, newLength));
          return newTab;
}
```

Application :

…

String[] oldTab = {"aaa","bbb","ccc","ddd"};

newTab = *AgrandirTailleTableau(oldTab, 5);*

System.out.println(newTab.getClass().getTypeName()); `java.lang.Object[]`

String[] ts = (String[]) newTab; ⟵ *Impossible : Object n'hérite pas de String*

Utilité des mécanismes réflexifs (2)

Exemple d'utilisation : Augmenter la taille d'un tableau

- Utilisation de *java.lang.reflect.Array* , *java.lang.Class et java.lang.Object*
- Array permet de créer des instances et de lire ou modifier les éléments

```
static Object AgrandirTailleTableau2(Object[] tab, int newLength) {
        Class type = tab.getClass().getComponentType();
        Object newTab = Array.newInstance(type, newLength);
         int oldLength = Array.getLength(tab);
        for (int i=0;i<Math.min(oldLength, newLength);i=i+1) {
                Array.set(newTab, i, tab[i]);
        }
        return newTab;
}
```

```
Object newTab = AgrandirTailleTableau2(oldTab, 5);
System.out.println(newTab.getClass().getTypeName());
String[] newTab2 = (String[]) newTab;
```

java.lang.String

java.lang.String[]

JUNIT3 : Un exemple de mise en œuvre de l'introspection

```
public class ExempleClasse {
  public static int addSiPetit (int a, int b) {
      ! Le résultat doit être inférieur à 100
    int res = a + b;
    if (res) > 100 res = -1
    return res;
    }
}
```

```
import junit.framework.*;
public class ExempleClasseTest extends TestCase {
  public void testAdditionnerSiPetit() throws Exception {
    assertEquals(2, ExempleClasse. additionnerSiPetit(1,1));
  } }
```

java -cp junit.jar;. junit.textui.TestRunner ExempleClasseTest

Exécution par introspection :

- c = Class.forName(« ExempleClasse » + « Test »)
- Vérifier qu'elle est sous classe de TestCase + créer une instance **i**
- Dans C.getDeclaredMethods() rechercher les méthodes de ExempleClasse qui contiennent « test » devant et les associer à « m »
- m.invoke(**i**)*

Autre approche possible ?

** **I** ou **null** : méthode d'instance ou de classe*

Liaison entre le moteur de test et les applications

```
164    public static void main(String[] args) {
165        System.out.println("1er argument:" + args[0]);
166        System.out.println("2eme argument: " + args[1]);
167        String packagePath = args[1];
168        String directoryContainingClassesToTest = args[0];
169        URL dirUrl;
170       try {
171           dirUrl = new URL("file://" + directoryContainingClassesToTest);
172           classLoader = new URLClassLoader(new URL[]{dirUrl});
173       } catch (MalformedURLException e) {
174           // TODO Auto-generated catch block
175           e.printStackTrace();
176       }
177        SmallJunitEngine ThisEngine = new SmallJunitEngine();
178        ThisEngine.initWithArgs(directoryContainingClassesToTest);
179        ThisEngine.engineIteration(packagePath);
180        System.out.println("Affichage du résultat des tests");
181        ThisEngine.journal.afficher();
182    }
```