

Introspection and reflexivity

Principles and application to Java

Philippe Collet

From a Michel Buffa's lecture

Master 1 IFI / International

2013-2014

<http://deptinfo.unice.fr/twiki/bin/view/Minfo/SoftEng1314>

Agenda

- RTTI in Java
- Reflexivity: definitions
- Analysing a class
- Analysing an object
- Example
- Handling methods
- Annotation
 - By default, examples are based on Java 2 / JDK 1.4.x
 - Zoom on extensions brought since Java 5 and 6

The class Class

RTTI in java

Run-Time Type Identification

- Java is maintaining what is called Run-Time Type Identification (RTTI) on all objects
- Enables one to know the *real* class of an object
- Enables the runtime system to implement dynamic binding (what method is really called)

```
GraphicObject o = new Circle();  
o.draw(); // from Circle or GraphicObject ?
```

The class `Class` and RTTI

- A class allows for accessing RTTI, the class `Class`

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

```
Person p1 = new Professor("Yoda")
```

```
Class c1 = p.getClass();
```

```
System.out.println(c1.getName() + " " +  
    p.getNom());
```

Displays

```
"Person Maurice"
```

```
"Professor Yoda" with p1...
```

The .class suffix

- An object of type Class can be obtained:

```
Class c11 = Professor.class;
```

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

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

...

- Useful to check the type with predefined types

Name = Class ?

- A class can be obtained from its name (~ functional languages)

```
String className =  
    "com.wars.star.Professor";  
Class c1 = Class.forName(className);
```

- **className** can be the name of an interface or a class
- Useful to **load classes** which name is not known in advance

Create instances without new

- Using the method `newInstance()` of the class **Class**

```
String className = "Professor";
```

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

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

- Note : it exists `newInstance(Object [] params)` of class `Constructor (java.lang.reflect)`

Methods of the class Class (Java 2)

```
String getName();
```

```
Class getSuperClass();
```

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

```
boolean isInterface();
```

```
String toString();
```

```
static Class forName(String name);
```

```
Object newInstance();
```

Methods of the class **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) ;
```

Reflexivity

Definition

- A « reflexive » system is able to represent itself
 - For computer languages, reflexivity is expressed as the capability for a language to describe aspects considered as implicit in the language
 - In the case of an object language:
 - Access to the class of an object : classes are objects
 - Instantiation of classes which name is only known at runtime
 - Access to attributes and methods of objects
 - Method invocation (message passing) with the method name only known at runtime

Meta-circular object schema



- Classes are instances of *Class* and thus are objects
 - Have attributes: attributes of the instances, their methods...
 - Have methods (message can be passed to them)
 - Can create instances
 - Enable to know value of attributes
 - Can call methods
- The Object class is a class, thus an instance of Class, it can then be handled in the same way

Reflexivity, introspection, meta-programming ?

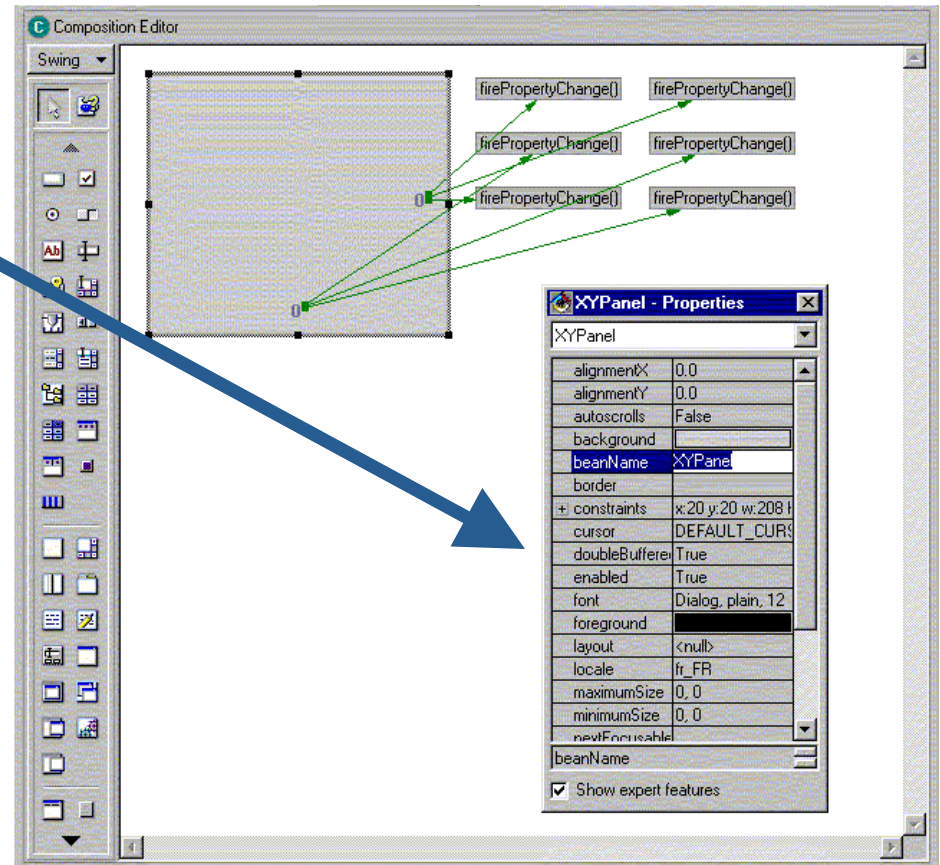
- Introspection
 - Know/inspect classes, objects, methods at runtime
 - Do it in the same language = reflexive introspection
- Meta-programming
 - Capability to modify language mechanisms using a META program
- Meta + reflex = inspect and modify behavior
- Java = reflexive introspection, no meta-programming protocol

How to do it in Java ?

- Class **Class** minimal in Java 1.0
- Improvement from version 1.1 due to Java Beans and RMI
 - Tools (RAD: JBuilder, Visual Café, Eclipse ...) need to deal with new classes at runtime (beans)
 - Classes are moved around in distributed architectures
- Such features are provided by the package **java.lang.reflect**
 - Analyse classes at runtime
 - Inspect objects at runtime
 - Write a generic toString() method
 - Handle generic arrays
 - Handle methods as pointers to function like in C/C++...

Reflexivity in a RAD tool

- Display of properties in an editing window
- Property = attribute with accessor and modifier (optional)
- Dynamic query on the class looking for **get...()** and **set...()**



Analysing a class

Analysing a class

- Three main classes in `java.lang.reflect`
 - `Field`, `Method`, `Constructor`
 - All have `getName()`
 - `Field` has `getType()` which returns an object of type `Class`
 - `Method` and `Constructor` have methods to get the return type and parameter types

Analysing a class (cont'd)

- These three classes have `getModifiers()` which returns an `int`, which bytes set to 0 or 1 mean static, public, private, etc...
 - Static methods from `java.lang.reflect.Modifier` are used to get these values
 - `Modifier` has methods such as `toString(int)`, `isFinal(int)`, `isPublic(int)`, `isPrivate(int)`

Example

- During the lab, a program will be able to produce, from a class name and a .class file:

```

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);
  ...
}
  
```

Methods of the class Class

- **Field[] getFields()**
 - Returns **only public attributes, local and inherited**
- **Field[] getDeclaredFields()**
 - Returns **all attributes local only**
- These two methods returns an array of null size if
 - No attributes
 - The class is actually a predefined type (int, double...)

Methods of Class (cont'd)

- **Method[] getMethods ()**
 - Returns **only public, local and inherited, methods**
- **Method[] getDeclaredMethods ()**
 - Returns **all, local only, methods**
- **Constructor[] getConstructors ()**
 - Returns all public constructors
- **Constructors[] getDeclaredConstructors ()**
 - Returns all constructors

Methods of Field, Method, Constructor

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

Lab : a class analyser

- Write a method
- **analyseClass (String className)**
- That displays all it is possible to obtain on a class which name is given as parameter

Analysing an object at runtime

Analysing an unknown object?

- Till now, we can
 - Determine **names** and **types** of the attributes of an object
 - Obtain from the object, its type (another object, of type **Class**)
 - Call **getDeclaredFields()** on the obtained object...
- We now want to obtain the **content** of an object, ie the content of its attributes
 - We don't know the object in advance, we don't know its class...

Accessing the value of an attribute

- Use the method `get()` of class `Field`
 - If `f` is an object of type `Field`
 - `f = field[i];`
 - If `obj` is an object of the class from which `f` is an attribute
 - Then `f.get(obj)` returns the value of the attribute `f` of the object `obj`

Example (in french ;-))

```
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;  
    }  
}
```

Example (cont'd)

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

...

```
Class c1 = p.getClass();
```

```
Field f = c1.getField("nom");
```

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

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

- But... This does not work!
 - As the attribute « nom » is **private** !
 - Raises an **IllegalAccessException**

Example (cont'd)

- The Java security manager enables one to get the attributes, but not always their value (think « password »)
- Possible solution:
 - Change the attribute to public (so so)
 - Put the code in the class `Personne` (so so)
 - Use **`AccessibleObject.setAccessible(AccessibleObject[] array, boolean flag)`**
 - Allows, for example, for giving access to private attributes
 - May also not work if the security manager did not give authorization (java.policy... cf. java rmi)
 - Then raises a **`SecurityException`**

Example (cont'd)

- What happens if the attribute is of a predefined type?
 - `f.get(obj)` returns an **Object** !
- In this case, it is the corresponding class which is chosen: **Integer**, **Double**, **Float**, etc...


```
Field f = cl.getField("age"); // age is an int
Object v = f.get(p);          // v is an Integer
```
- With JDK 1.3 et 1.4 :
 - `f.getInt()`, `f.getDouble()`, etc...
- With Java 5 and on:
 - Auto-boxing simplifies the notation

Lab: a generic toString method

- Quite often, the toString method is redefined to describe an object by displaying values of its attributes.
- With reflexivity, it is possible to:
 - Write a method once
 - Place it in the classes source or quite high in a class hierarchy
 - The code is thus written once!

Usage example: a growing array

The Array class from java.lang.reflect

- Common issue: a array of objects
 - Of a given type
 - And the array is full
- We can make it grow!

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

```
...
```

```
// the array is full
```

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

How to do it?

- Let's try this:

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

- Problem: the type **Object[]** can not here be *typecasted* in **Personne[]**
- Why?

Methods of Array and Class classes

- InArray
 - `static Object newInstance(Class componentType, int length)`
 - `static int getLength()`
 - `boolean isArray()`
- In Class
 - `Class getComponentType()` , apply only on an [Array](#)
 - Ex: `Class type =
o.getClass().getComponentType();`
 - `o` must be anArray, and `type` represents the type of the array elements.

Lab: modify the preceding code

- Solution: use reflexivity to create a new array of the same type as the original array
- During the lab, modify **Object[] growArray()**
- ... so that it works!
 - Think at using methods from **Array** and **Class**
 - Test with **int[]** ! Trap ahead!

Pointers on functions?

Pass a parameter to a function, of type Method ?

- Emulate some pointers to functions ?

```

public print(double start, double end, double step,
  Method f) {
  for(double x = start; x < end; x+= step) {
    f(x);
  }
}
  
```

- Not so easy!

Calling a method

- The class Method has

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

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

- `o` is the object which is going to receive the message
 - For a static method, `o` is `null`
- `args` is the parameter list

- Example

- To emulate `yoda.getNom()` where `yoda` is an instance of `Personne` and `f` represents `getNom()`

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


Calling a method (cont'd)

- With predefined types, use « wrapper classes"
Integer, **Float**, **Double**, etc...
- Example with **f** that represents **setAge(int age)** of the class **Personne**

```
// JDK 1.2 to 1.4  
Object[] args = {new Integer(735)};  
f.invoke(yoda, args);
```

```
// Java 5 : Autoboxing + variable argument list  
f.invoke(yoda, 735);
```

How to obtain an object of type Method ?

- Use `getMethod(...)` or `getDeclaredMethods()` of class `Class`
- Due to overloading, one must be able to precisely types of parameters with **`getMethod(...)`**

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

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

- Example

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

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

```
// Java 5
```

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

Other features

- Class Proxy
 - Static methods to dynamically create classes and instances *proxy*
- A proxy class (dynamic) is a class that implements a list of interfaces, specified at runtime
 - With no existing **text** class
- Reflexivity allows for creating new instances
- Pros?
 - Create instances compatible with several interfaces (easier for adaptation), with no class writing beforehand

New reflexive features in Java 5

- Metadata...
 - Annotations @... In the Java code
 - Content of the annotations accessible by introspection at runtime
- Genericity is handled
 - Class (concept of type and paramterized type)
- Autoboxing of base types
 - Eases the extraction of values from these types

Annotations in Java

- An annotation allows for marking some elements of the Java language so to add some particularities.
- These annotations can then be used at compile or execution time so to automate some tasks

@MyAnnotation

```

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

- *Standards* annotations : @Deprecated, @Override, @SuppressWarnings("deprecation")
- Meta-annotations (to annotate other annotations) :
 - @Documented, @Inherit
 - @Retention(RetentionPolicy.SOURCE ou .CLASS ou .RUNTIME)
 - ...

Creation of an annotation

- Creation

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

```
@Documented
```

```
@Retention(SOURCE)
```

```
public @interface TODO {
```

```
    /* Message describing the task */
```

```
    String value(); Attribute (limited type)
```

```
}
```

- Usage

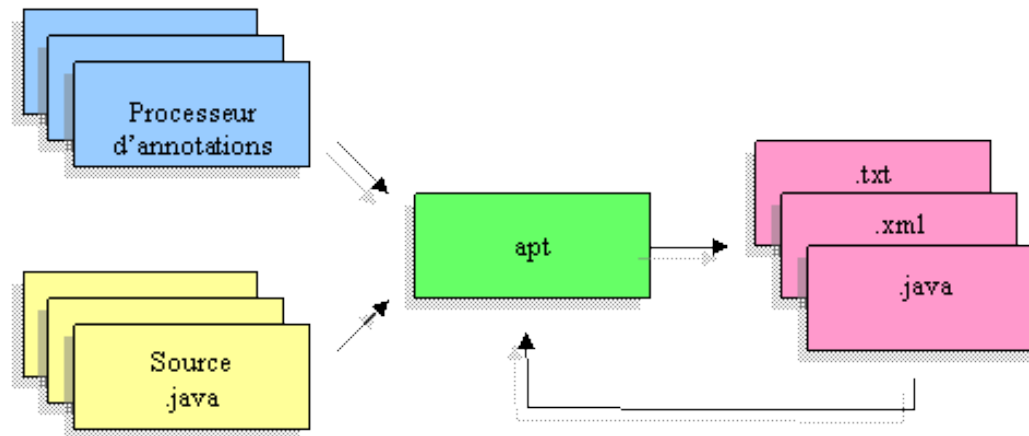
```
@TODO(value="Code this ASAP...")
```

```
    public void ZeMethod () {
```

```
        ...
```

Using annotations

- apt (annotation processing tool), Java 5



© jmdoudoux

- Introspection : `getAnnotations` on all elements : Class, Method, etc.
- Java (Java 6) compiler: API Pluggable Annotation Processing
 - External tools are not needed anymore

