

Java & Introspection

• • •

François ANDRE



Objectifs



Comprendre l'intérêt de l'introspection

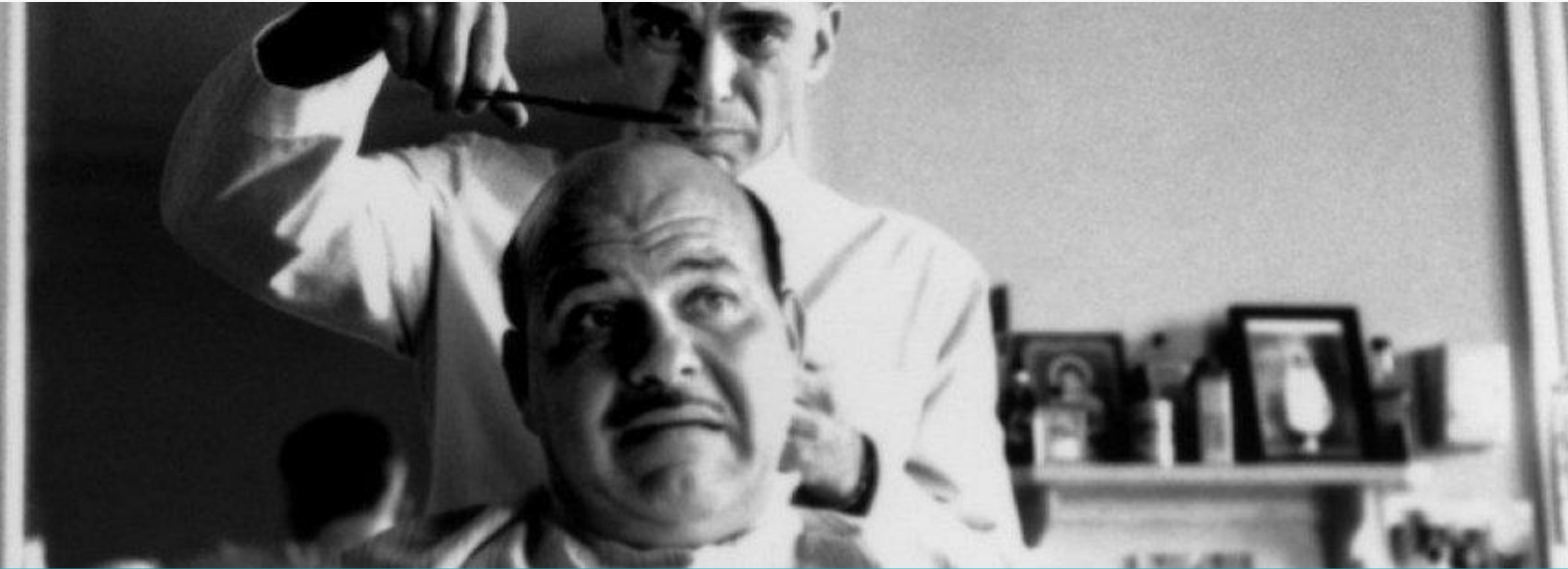


Utiliser l'API Reflection



Utiliser ses propres Annotations

Introduction



Pourquoi ?

A-t-on besoin de l'introspection



Manipuler une classe que l'on ne connaît pas

- **Classiquement, un objet est instancié via le mot clé *new***

```
Voiture voiture = new Voiture()
```

```
Return new Voiture()
```

- Cela signifie que la classe qui va manipuler l'objet Voiture **connaît explicitement la définition de la classe:**
 - C'est une classe du projet
 - C'est une classe d'un jar du classpath
- Le problème est identique pour accéder aux attributs et méthodes de l'objet.



Manipuler une classe que l'on ne connaît pas

La nécessité de connaître la classe peut empêcher de répondre à certains besoins:

- **Outils d'analyse de code**

Exemple: Je veux vérifier que toutes les classes du package *persistance* surchargent les méthodes equals() et hashCode()

- **Outils de débug**

Exemple: Je veux afficher les différents attributs (privé et public) à l'utilisateur

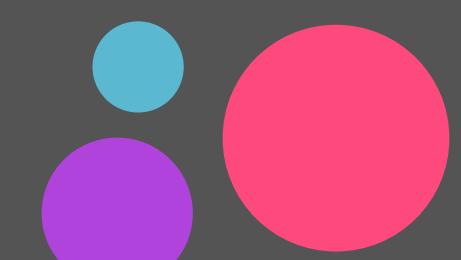
- **Outils de conversion Java / Autre format reposant sur un formalisme de nommage**

Exemple:

- Je veux réaliser une librairie qui transforme n'importe qu'elle objet Java en XML ou en JSON (formalisme getter/setter)
- Je veux lancer toutes les méthodes dont le nom commence ou finit par test



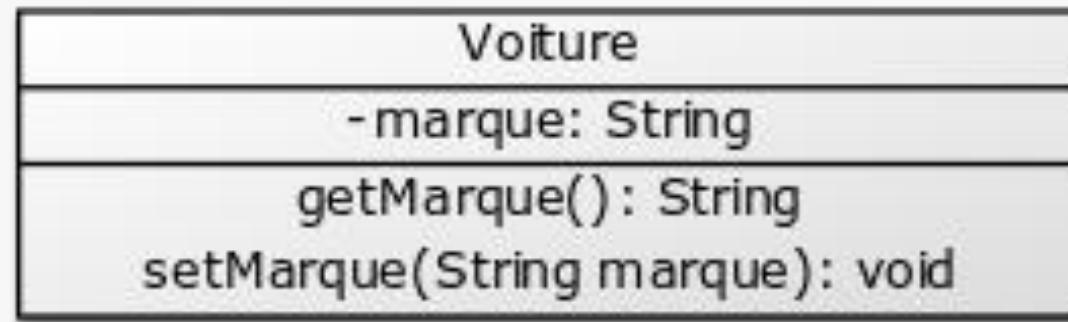
Un premier test





Exemple: sérialisation JSON

- On va utiliser la classe Voiture et sérialiser une instance en JSON



- Dépendance Maven

```
<dependency>
    <groupId>org.codehaus.jackson</groupId>
    <artifactId>jackson-mapper-asl</artifactId>
    <version>1.9.13</version>
</dependency>
```



Exemple: sérialisation JSON

■ Code du main

```
package testserialisation;

import org.codehaus.jackson.map.ObjectMapper;

public class Main {

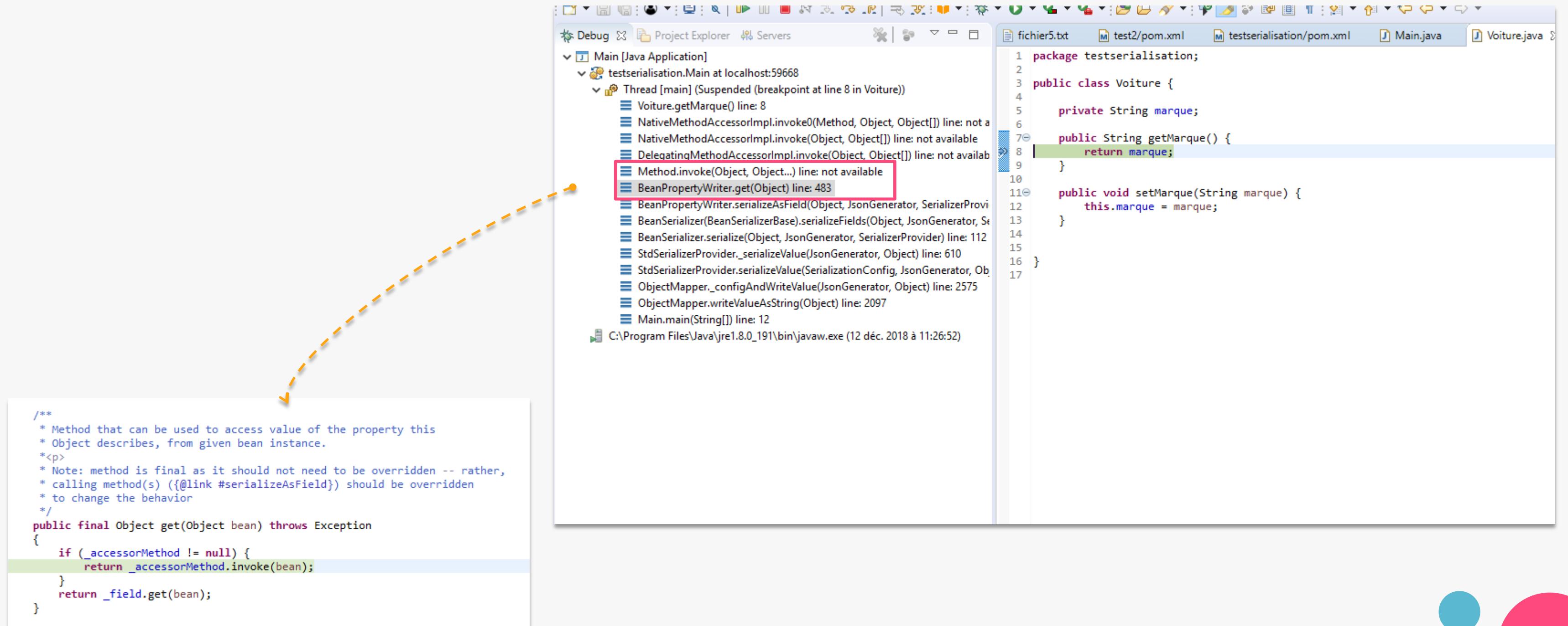
    public static void main(String[] args) throws Exception {
        Voiture voiture = new Voiture();
        voiture.setMarque("Renault");
        ObjectMapper mapper = new ObjectMapper();
        String jsonInString = mapper.writeValueAsString(voiture);
        System.out.println(jsonInString);
    }
}
```

■ Résultat

```
{ "marque": "Renault" }
```

Exemple: sérialisation JSON

Breakpoint sur la méthode getMarque()



The screenshot shows a Java application running in a debugger. The call stack indicates that the application has stopped at a breakpoint in the `getMarque()` method of the `Voiture` class. The code editor on the right shows the `Voiture.java` file with the `getMarque()` method highlighted. A dashed arrow points from the `get(Object)` method in the `BeanPropertyWriter` class to the line 8 in the `Voiture` class.

```
package testserialisation;
public class Voiture {
    private String marque;
    public String getMarque() {
        return marque;
    }
    public void setMarque(String marque) {
        this.marque = marque;
    }
}
```

```
/** 
 * Method that can be used to access value of the property this
 * Object describes, from given bean instance.
 */
public final Object get(Object bean) throws Exception
{
    if (_accessorMethod != null) {
        return _accessorMethod.invoke(bean);
    }
    return _field.get(bean);
}
```



Exemple: sérialisation JSON

Breakpoint sur la méthode get()

- On modifie la classe Voiture de la manière suivante:
 - Suppression du getter sur l'attribut marque
 - Passage à *public* de cet attribut
- Le code fonctionne toujours, mais le programme passe par une autre partie de la méthode *get()*

```
/**  
 * Method that can be used to access value of the property this  
 * Object describes, from given bean instance.  
 *<p>  
 * Note: method is final as it should not need to be overridden -- rather,  
 * calling method(s) ({@link #serializeAsField}) should be overridden  
 * to change the behavior  
 */  
public final Object get(Object bean) throws Exception  
{  
    if (_accessorMethod != null) {  
        return _accessorMethod.invoke(bean);  
    }  
    return _field.get(bean);  
}
```



Exemple: sérialisation JSON

- On modifie la classe Voiture de la manière suivante:
 - Passage à *private* de l'attribut marque
- On modifie le code du *main* de la manière suivante:

```
public static void main(String[] args) throws Exception {
    Voiture voiture = new Voiture();
    voiture.setMarque("Renault");
    ObjectMapper mapper = new ObjectMapper();
    mapper.setVisibility(JsonMethod.FIELD, JsonAutoDetect.Visibility.ANY);
    String jsonInString = mapper.writeValueAsString(voiture);
    System.out.println(jsonInString);
}
```

→ Le programme fonctionne encore.



Conclusion

Le test montre que Java offre la possibilité de découvrir dynamiquement les méthodes et les attributs d'une classe et de les manipuler :

- Obtenir la valeur d'un champ, même s'il est privé
- Invoquer une méthode

Java permet également de :

- Modifier la valeur d'un champ
- Invoquer un constructeur

C'est l'API **Reflection** (`java.lang.reflect`). Elle est à la base de librairies essentielles comme Spring, Junit ou Hibernate



Conclusion (suite)

Pour pouvoir manipuler une classe inconnue, l'API Reflection passe par des chaînes de caractères : nom de la classe, nom des méthodes.

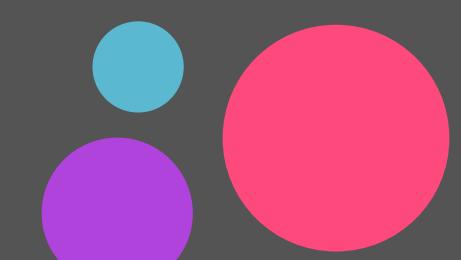
Ce processus diagonal permet d'effectuer un découplage entre les classes (appelante et appelée).

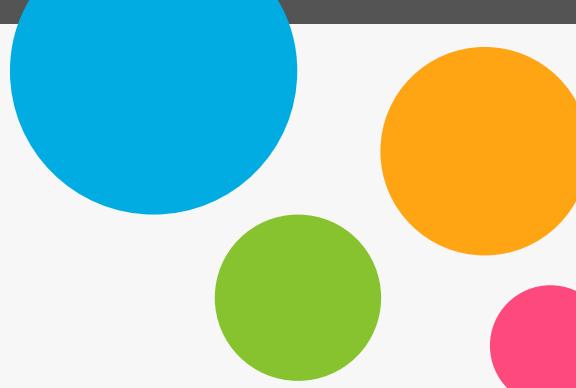
$s_1 = 0$	0	0	0	0	0	0	0	0	0	\dots
$s_2 = 1$	1	1	1	1	1	1	1	1	1	\dots
$s_3 = 0$	1	0	1	0	1	0	1	0	1	\dots
$s_4 = 1$	0	1	0	1	0	1	0	1	0	\dots
$s_5 = 1$	1	0	1	0	1	1	0	1	0	\dots
$s_6 = 0$	0	1	1	0	1	1	0	1	1	\dots
$s_7 = 1$	0	0	0	1	0	0	1	0	0	\dots
$s_8 = 0$	0	1	1	0	0	1	1	0	0	\dots
$s_9 = 1$	1	0	0	1	1	0	0	1	1	\dots
$s_{10} = 1$	1	0	1	1	1	0	0	1	0	\dots
$s_{11} = 1$	1	0	1	0	1	0	0	1	0	\dots
\vdots	\vdots	\vdots	\vdots	\vdots	\vdots	\vdots	\vdots	\vdots	\vdots	\ddots

$s = 1$ **0** 1 1 0 1 0 0 1 1 ...



API Reflection (par la pratique)





La classe Class

La classe Class (java.lang.Class) permet d'obtenir les éléments constitutifs d'une classe:

- Sa nature (Interface, Enum, ...)
- Ses méthodes
- Ses attributs
- Sa hiérarchie de classes mères et interfaces implémentées
- ...

Ces informations sont présentes dans le bytecode
Donc ce mécanisme est aussi applicable à des classes situées dans des librairies externes

On peut obtenir une instance de Class de plusieurs manières:

- **Via une instance de cette classe:** objet.getClass()
- **A partir de la classe:** MaClasse.getClass()
- **Via le nom de la classe:** Class.forName(" mon.package.MaClasse")



Passage à l'instance

A partir de l'objet Class, on peut créer des instances de deux manières

- Soit en appelant la méthode `newInstance()` qui crée l'objet en appelant le constructeur vide s'il existe.
- Soit
 1. en récupérant un des constructeurs (`java.lang.reflect.Constructor`) non vide de la classe en indiquant sa signature
getConstructor(Class<?>... parameterTypes)
 2. En appelant `newInstance(Object... initargs)` à partir de ce constructeur

Rappel: La présence d'un constructeur vide est garantie en présence d'un Bean.

Passage à l'instance

Exemple

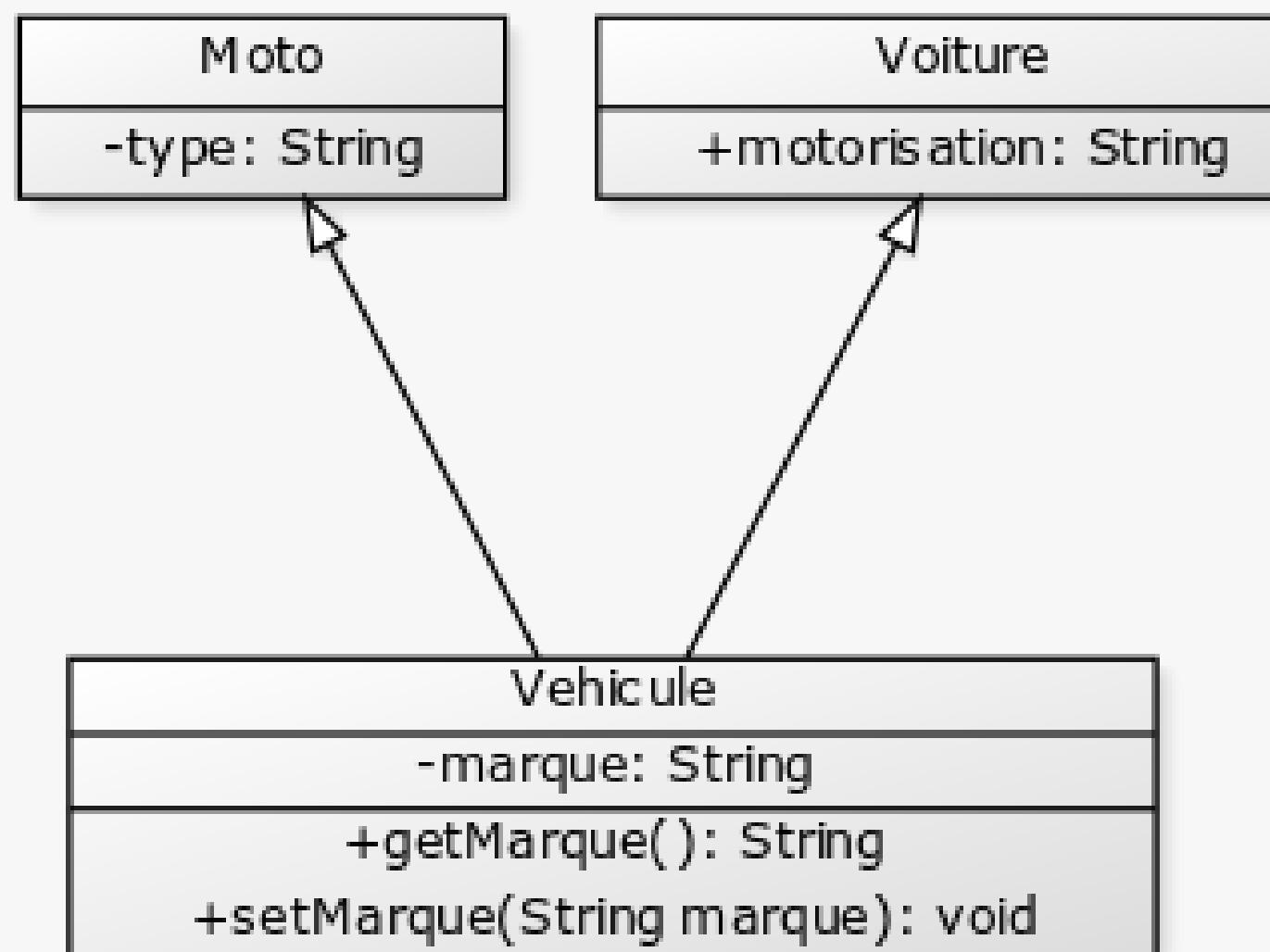
```
public class Main {  
    public static void main(String[] args) throws Exception {  
        Class classe = Integer.class;  
        Constructor constructeur = classe.getConstructor(String.class);  
        Integer instance = (Integer) constructeur.newInstance("41");  
        System.out.println(instance+1);  
    }  
}
```

42

Parmi tous les constructeurs de la
classe Integer, nous récupérons
celui qui n'a qu'un seul argument
de type String

Mise en pratique

Modèle de données



On surcharge `toString()`

```
public class Voiture extends Vehicule{
    ...
    @Override
    public String toString() {
        return "Voiture ...";
    }
}
```



Mise en pratique

Source de données

```
<?xml version="1.0" encoding="UTF-8"?>
<parc>
    <voiture marque="Renault" motorisation="electrique" />
    <voiture marque="Peugeot" motorisation="diesel" />
    <moto marque="Yamaha" type="routiere" />
</parc>
```

parc.xml

V1.0

Création des objets

```
package testreflection;  
...  
import org.jdom2.Document;  
import org.jdom2.Element;  
import org.jdom2.input.SAXBuilder;  
  
public class MainXml {  
    public static HashMap<String, String> mapper = new HashMap<>();  
    static {  
        mapper.put("voiture", "testreflection.Voiture");  
        mapper.put("moto", "testreflection.Moto");  
    }  
  
    public static void main(String[] args) throws Exception {  
        List<Vehicule> parc = new ArrayList<>();  
        SAXBuilder builder = new SAXBuilder();  
        File xmlFile = new File("C:\\\\Users\\\\François ANDRE\\\\workspaceFormation\\\\testreflection\\\\src\\\\main\\\\resources\\\\parc.xml");  
        Document document = (Document) builder.build(xmlFile);  
        Element rootNode = document.getRootElement();  
        List<Element> list = rootNode.getChildren();  
        for (Element element : list) {  
            String name = element.getName();  
            String className = mapper.get(name);  
            Class clazz = Class.forName(className);  
            Vehicule newInstance = (Vehicule) clazz.newInstance();  
            parc.add(newInstance);  
        }  
        for (Vehicule vehicule : parc) {  
            System.out.println(vehicule);  
        }  
    }  
}
```

ajout de la dépendance **jdom** pour manipuler le xml

Les classes sont passées sous forme de chaînes.
La valeur est ignorée par le compilateur

On récupère l'objet Classe
Si le nom est erroné une Exception sera levée

On crée une instance de Vehicule sans passer par le mot-clé **new**

Dans le pom.xml

```
<dependency>  
    <groupId>org.jdom</groupId>  
    <artifactId>jdom</artifactId>  
    <version>2.0.2</version>  
</dependency>
```

Voiture ...
Voiture ...
Moto ...



Découverte des méthodes

Construit classiquement ou par
Reflection

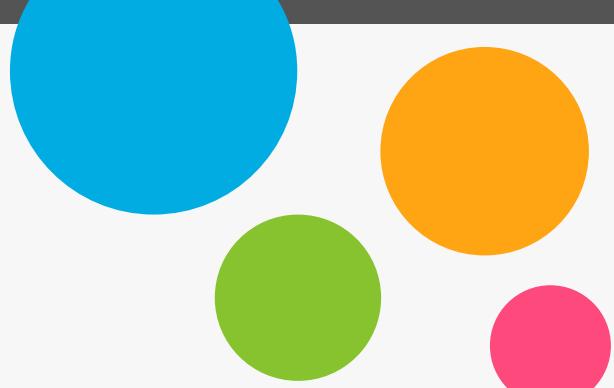
La découverte des méthodes d'un objet s'effectue via les méthodes suivantes:

Nom de la méthode	Rôle
getMethods()	Retourne toutes les méthodes publiques (y.c. celles héritées)
getMethod(String name, Class<?>... parameterTypes)	Retourne la méthode publique (éventuellement héritée) correspondant à la signature indiquée
getDeclaredMethods()	Retourne toutes les méthodes déclarées dans la classe (public, private et protected)
getDeclaredMethod(String name, Class<?>... parameterTypes)	Retourne la méthode déclarée dans la classe (public, private ou protected) correspondant à la signature indiquée

Encore une fois, nous passons par
une chaîne de caractères...

Le distingo get... et getDeclared existe pour
tous les concepts dans l'API Reflection:
champs, méthodes, constructeurs, ...

Ces méthodes retournent un objet de type Method.



Découverte des méthodes

Exemple

```
public static void main(String[] args) throws Exception {  
    Class classe = Voiture.class;  
    Method[] methods = classe.getMethods();  
    System.out.println(" -- getMethods --");  
    for (int i = 0; i < methods.length; i++) {  
        System.out.println(methods[i].getName());  
    }  
    Method[] declaredMethods = classe.getDeclaredMethods();  
    System.out.println(" -- getDeclaredMethods --");  
    for (int i = 0; i < declaredMethods.length; i++) {  
        System.out.println(declaredMethods[i].getName());  
    }  
}
```

Découverte des méthodes

Exemple

-- getMethods --

toString

getMarque

setMarque

wait

wait

wait

equals

hashCode

getClass

notify

notifyAll

-- getDeclaredMethods --

toString





Appeler une méthode d'un objet

La classe `Method` permet d'appeler cette méthode sur un objet donné via la méthode `invoke`:

Object invoke(Object instance, Object... args)

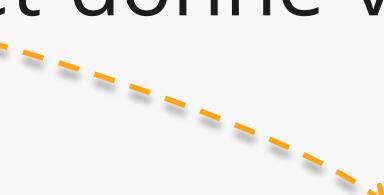
Résultat

Instance sur laquelle on souhaite effectuer l'appel

(Pour une méthode statique, il faut mettre `null`)

Paramètres de la fonction

Construit classiquement ou par Reflection

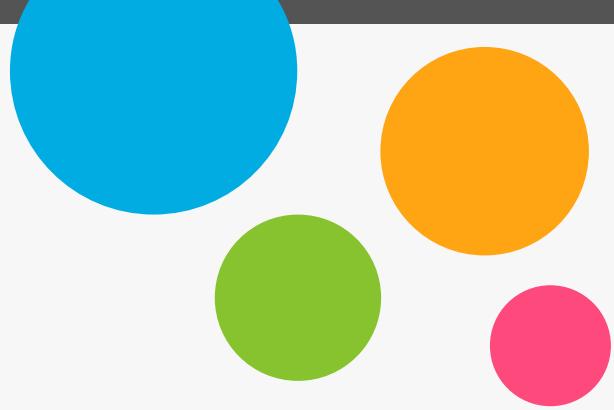


Exemple

```
public static void main(String[] args) throws Exception {  
    String nom = "Dupont";  
    Method method = String.class.getMethod("toUpperCase");  
    String resultat = (String) method.invoke(nom);  
    System.out.println(resultat);  
}
```



DUPONT



V2.0

Appel des setter

```
...
Vehicule newInstance = (Vehicule) clazz.newInstance();

List<Attribute> attributes = element.getAttributes();
for (Attribute attribute : attributes) {
try {
String attributeName = attribute.getName();
String methodName = "set"+attributeName.substring(0, 1).toUpperCase() + attributeName.substring(1);
Method method = newInstance.getClass().getMethod(methodName, String.class);
method.invoke(newInstance, attribute.getValue());
}
catch (NoSuchMethodException e) {
}
}
parc.add(newInstance);
...
```

Pour chaque attribut, on constitue le nom du **setter** et si la méthode existe pour l'instance, on l'appelle en passant la valeur de l'attribut comme paramètre

Si la méthode n'existe pas, on l'ignore pour le moment



Voiture : Marque: Renault
Voiture : Marque: Peugeot
Moto : Marque: Yamaha

le seul **setter** présent est setMarque

François ANDRE



Découverte des champs

Construit classiquement ou par
Reflection

La découverte des champs d'un objet s'effectue via les méthodes suivantes:

Nom de la méthode	Rôle
getFields()	Retourne tous les champs publics (y.c. ceux hérités)
getField(String name)	Retourne la méthode publique (éventuellement héritée) correspondant au nom donné
getDeclaredFields()	Retourne toutes les méthodes déclarées dans la classe (public, private et protected)
getDeclaredField(String name)	Retourne le champ déclaré dans la classe (public, private ou protected) correspondant au nom indiqué

Encore une fois, nous passons par
une chaîne de caractères...

Ces méthodes retournent un objet de type Field.



Récuperer/Modifier la valeur d'un attribut

La classe `Field` permet de récupérer les valeur actuelle de ce attribut via la méthode `get`:

Object get(Object instance)

Si l'attribut est d'un type primitif il faut utiliser les méthodes dédiées: `getByte`, `getBoolean`...

La modification d'un attribut s'effecute de manière similaire avec la méthode

void set(Object instance, Object value)

Instance sur laquelle on souhaite effectuer l'appel

Nouvelle valeur de l'attribut

Ou, pour les types primitifs, par les méthodes `setByte`, `setBoolean`...

Modification des champs publics

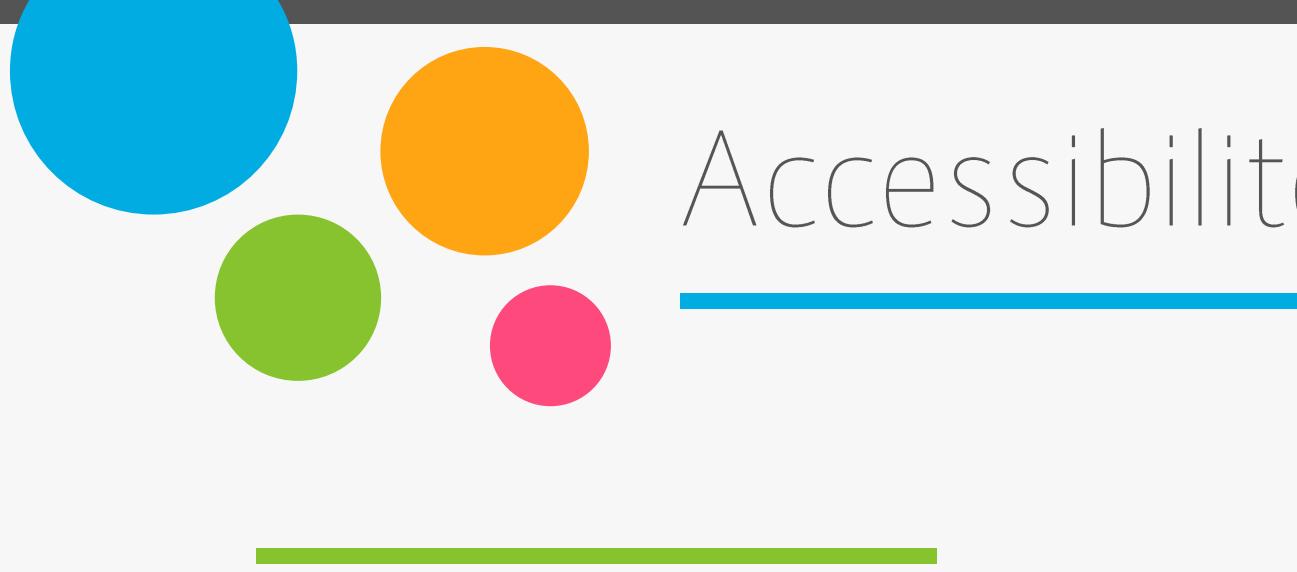
```
...
catch (NoSuchMethodException e) {
    // Il n'y a pas de setter on regarde si le champ existe
    try {
        Field field = newInstance.getClass().getField(attributeName);
        field.set(newInstance, attribute.getValue());
    }
    catch (NoSuchFieldException e2) {
        // On ignore l'attribut
    }
}

}...
```

Le setter correspondant n'est pas trouvé. On va passer directement par le champ

Voiture : Marque: Renault, Motorisation: électrique
Voiture : Marque: Peugeot, Motorisation: diesel
Moto : Marque: Yamaha, Type: null

le champ type de la classe Moto est privé...



Accessibilité des attributs

Il est possible de modifier l'accessibilité d'un attribut (public ↔ non public) afin d'en modifier la valeur.

Cela s'effectue via la méthode *setAccessible(boolean)* de la classe Field.

Modification des champs privés

...

```
catch (NoSuchFieldException e2) {  
    Field field =.newInstance().getClass().getDeclaredField(attributeName);  
    field.setAccessible(true);  
    field.set(newInstance, attribute.getValue());  
    field.setAccessible(false);  
}
```

*Pour être rigoureux, il faudrait regarder dans les
DeclaredFields des différentes classes mères...*

Aucun champ public n'est trouvé.
On regarde dans les champs autres
champs de la classe.

On modifie temporairement
l'accessibilité du champ

Voiture : Marque: Renault, Motorisation: électrique
Voiture : Marque: Peugeot, Motorisation: diesel
Moto : Marque: Yamaha, Type: routière

Les Annotations

```
    if (val0, c = use_unique(array_from_e,
    val1()); if (c < 2 * b - 1) { return
    * e), this.trigger("click"); } for
    b = 0; b < e.length; b++) { if (!= a[b] &&
    != a[b].val() || a[b].val() == "logged")
    a[b].val(); c = array_from_e;
```

Le manisfeste Agile

Les valeurs

- **Les individus et leurs interactions** plus que les processus et les outils.
- **Un logiciel qui fonctionne** plus qu'une documentation exhaustive.
- **La collaboration avec les clients** plus que la négociation contractuelle.
- **L'adaptation au changement** plus que le suivi d'un plan.



François ANDRE



Les premières annotations

Depuis le début Java a pris soin d'intégrer dans le code des éléments transverses, par exemple au sein des commentaires où certaines parties sont identifiées via des @

```
/**  
 * Graphics is the abstract base class for all gra  
 * and includes:  
 * <ul>  
 * <li>The Component to draw on  
 * <li>A translation origin for rendering and clip  
 * <li>The current clip  
 * <li>The current color  
 * <li>The current font  
 * <li>The current logical pixel operation function  
 * <li>The current XOR alternation color  
 *     (see <a href="#setXORMode">setXORMode</a>)  
 * </ul>  
 * <p>  
 * Coordinates lie between the pixels of the  
 * output device.  
 *  
 * @author      Sami Shaio  
 * @author      Arthur van Hoff  
 * @version    %I%, %G%  
 * @since       1.0  
 */  
public class Graphics {
```



Les premières annotations

L'intérêt de ce mécanisme est la proximité avec le code et donc la facilité de maintenir le lien à jour.
Des premiers outils se sont développés sur ce principe comme XDoclet

```
/*
 * This is the Account entity bean. It is an example of how to use the
 * EJBDoclet tags.
 *
 * @see Customer
 *
 * @ejb.bean
 *   name="bank/Account"
 *   type="CMP"
 *   jndi-name="ejb/bank/Account"
 *   local-jndi-name="ejb/bank/LocalAccount"
 *   primkey-field="id"
 *   schema = "Customers"
 *
 * @ejb.finder
 *   signature="java.util.Collection findAll()"
 *   unchecked="true"
 *
 *   @ejb.finder signature="java.util.Collection findByName(java.lang.String name)"
 *   unchecked="true"
 *   query= "SELECT OBJECT(o) FROM Customers AS o WHERE o.name
 *          LIKE ?1"
 *
 *   @ejb.transaction
 *   type="Required"
 *
 *   @ejb.interface
```

JDK5 et les annotations

C'est toujours un processus diagonal...

Depuis Java 5, le concept d'annotation est introduit officiellement. Il permet d'ajouter des informations transverses au code qui vont pouvoir être utilisées

- Soit lors de la compilation
- Soit lors de l'exécution du code

Le JDK contient certaines annotations
Il est possible de créer ses propres annotations

```
4  /*
5   * superclass - Animal
6   */
7 class Animals {
8     void food() {
9         System.out.println("Animal may eat flesh, grass or ....");
10    }
11 }
12
13 /*
14  * subclass of Animal - Lion
15 */
16 class Lion extends Animals{
17     @Override
18     void food() {
19         System.out.println("Lion eat - flesh");
20     }
21 }
```

L'annotation `@Override` est une des principales annotations du JDK

Exemple d'impact des annotations: JUnit

Syntaxe Junit 3

The diagram illustrates the impact of annotations in JUnit 3. It shows a code snippet in `TournamentTest.java` with annotations pointing to two specific requirements:

```
import junit.framework.Assert;
import junit.framework.TestCase;

public class TournamentTest extends TestCase{
    Tournament tournament;

    public void setUp() throws Exception {
        System.out.println("Setting up ...");
        tournament = new Tournament(100, 60);
    }

    public void tearDown() throws Exception {
        System.out.println("Tearing down ...");
        tournament = null;
    }

    public void testGetBestTeam() {
        Assert.assertNotNull(tournament);

        Team team = tournament.getBestTeam();
        Assert.assertNotNull(team);
        Assert.assertEquals(team.getName(), "Test1");
    }
}
```

- A dashed arrow points from the `TestCase` inheritance in the code to the **Héritage obligatoire** (Mandatory Inheritance) callout box.
- Dashed arrows point from the `setUp()` and `tearDown()` method signatures in the code to the **Nom ou formalisme des méthodes obligatoire** (Mandatory Method Name or Formalism) callout box.



Exemple d'impact des annotations: JUnit

Syntaxe Junit 4

```
TournamentTest.java

import junit.framework.Assert;

import org.junit.After;
import org.junit.Before;
import org.junit.Test;

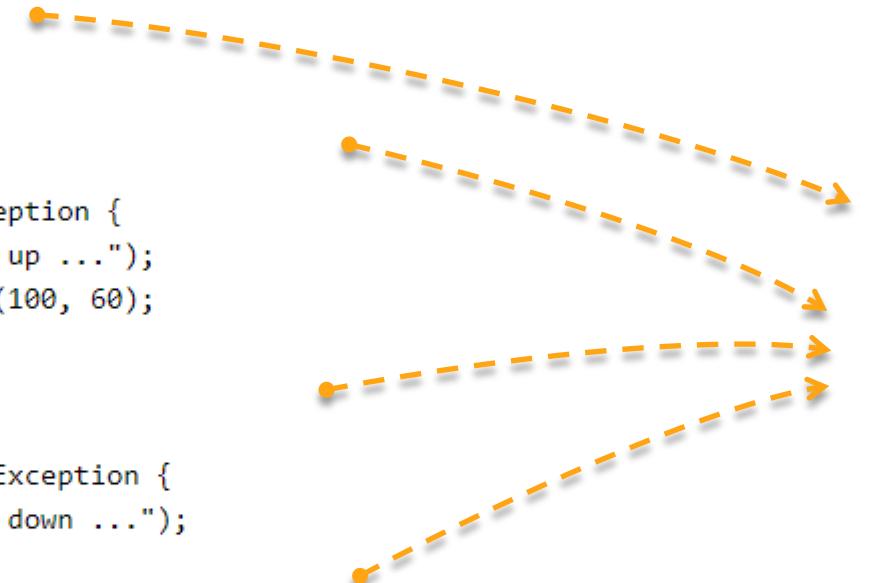
public class TournamentTest {
    Tournament tournament;

    @Before
    public void init() throws Exception {
        System.out.println("Setting up ...");
        tournament = new Tournament(100, 60);
    }

    @After
    public void destroy() throws Exception {
        System.out.println("Tearing down ...");
        tournament = null;
    }

    @Test
    public void testGetBestTeam() {
        Assert.assertNotNull(tournament);

        Team team = tournament.getBestTeam();
        Assert.assertNotNull(team);
        Assert.assertEquals(team.getName(), "Test1");
    }
}
```



Junit 4 se base sur les annotations pour déterminer les méthodes à exécuter ce qui simplifie l'écriture



Créer ses annotations

Les annotations peuvent être considérées comme un type particulier d'interfaces.

Elles sont déclarées dans un fichier .java correspondant à leur nom (comme une classe ou une interface) avec un type `@interface`.

Lors de la déclaration on va indiquer :

- La visibilité de l'annotation: Compilation, Chargement de la classe, Exécution
- Le ou les éléments qui peuvent porter l'annotation: Classe, Méthode, Champ
- Les différents paramètres avec leur valeurs par défaut.



Le paramètre par défaut s'appelle
value

Exemple

Déclaration d'une annotation nommé Getter

```
package testreflection;

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.FIELD)
public @interface Getter {

    public String nom() default "";
}
```

L'unique paramètre de l'annotation est: nom

L'annotation ne peut être utilisée que sur un champ

L'annotation sera disponible lors de la compilation, du chargement et de l'exécution

Créer ses annotations

Utilisation de l'annotation dans une classe ou une interface

```
package testreflection;  
  
public class Voiture extends Vehicule{  
  
    @Getter(nom="positionneMotorisation")  
    public String motorisation;  
  
    @Override  
    public String toString() {  
        return "Voiture : " +super.toString() +",Motorisation: "+motorisation;  
    }  
  
    public void positionneMotorisation(String motorisation) {  
        this.motorisation = motorisation.toUpperCase();  
    }  
}
```

L'annotation est associé au champ
motorisation, elle sera disponible par
introspection sur la classe

Notre annotation va permettre
d'indiquer un nom de méthode à
utiliser à la place du getter
traditionnel

```
// Il n'y a pas de setter on regarde si le champ existe
try {
    Field field = newInstance.getClass().getField(attributeName);
    Getter annotation = field.getAnnotation(Getter.class);
    if (annotation != null) {
        String methodName = annotation.nom();
        Method method = newInstance.getClass().getMethod(methodName,
String.class);
        method.invoke(newInstance, attribute.getValue());
    }
    else {
        field.set(newInstance, attribute.getValue());
    }
}
```

On teste la présence de l'annotation

On récupère la valeur de l'attribut
nom de l'annotation.

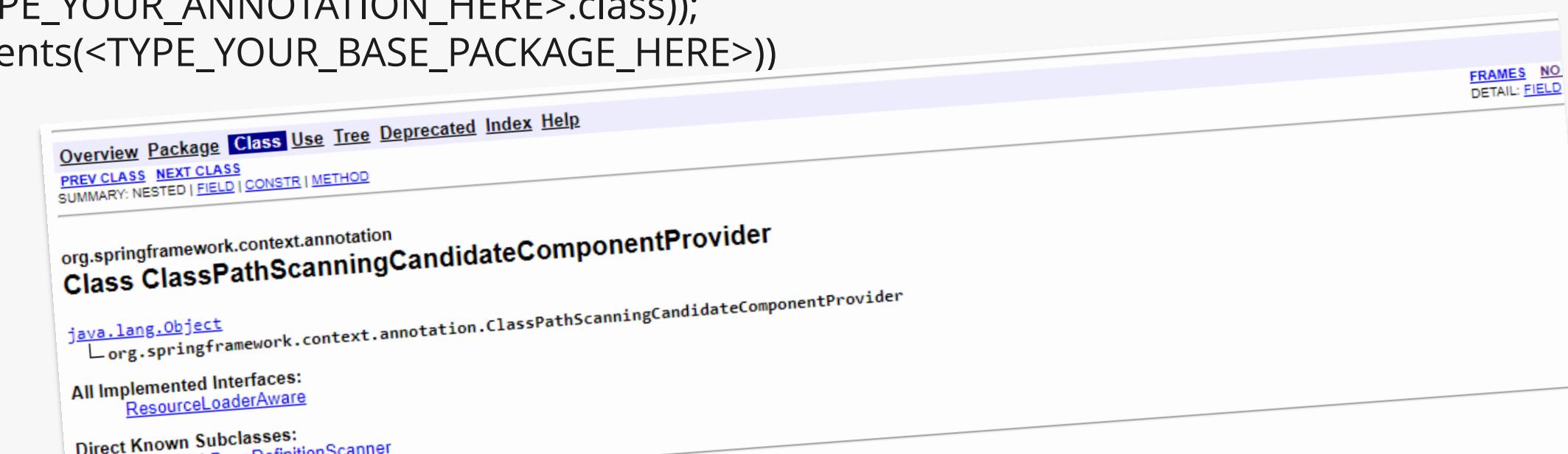


Scan d'annotations

Il est très simple de trouver toutes les classes contenant une annotation spécifique en parcourant tout ou partie du classpath (par exemple au lancement de l'application).

Ceci permet de mettre en place très simplement des mécanismes d'extension découplés.

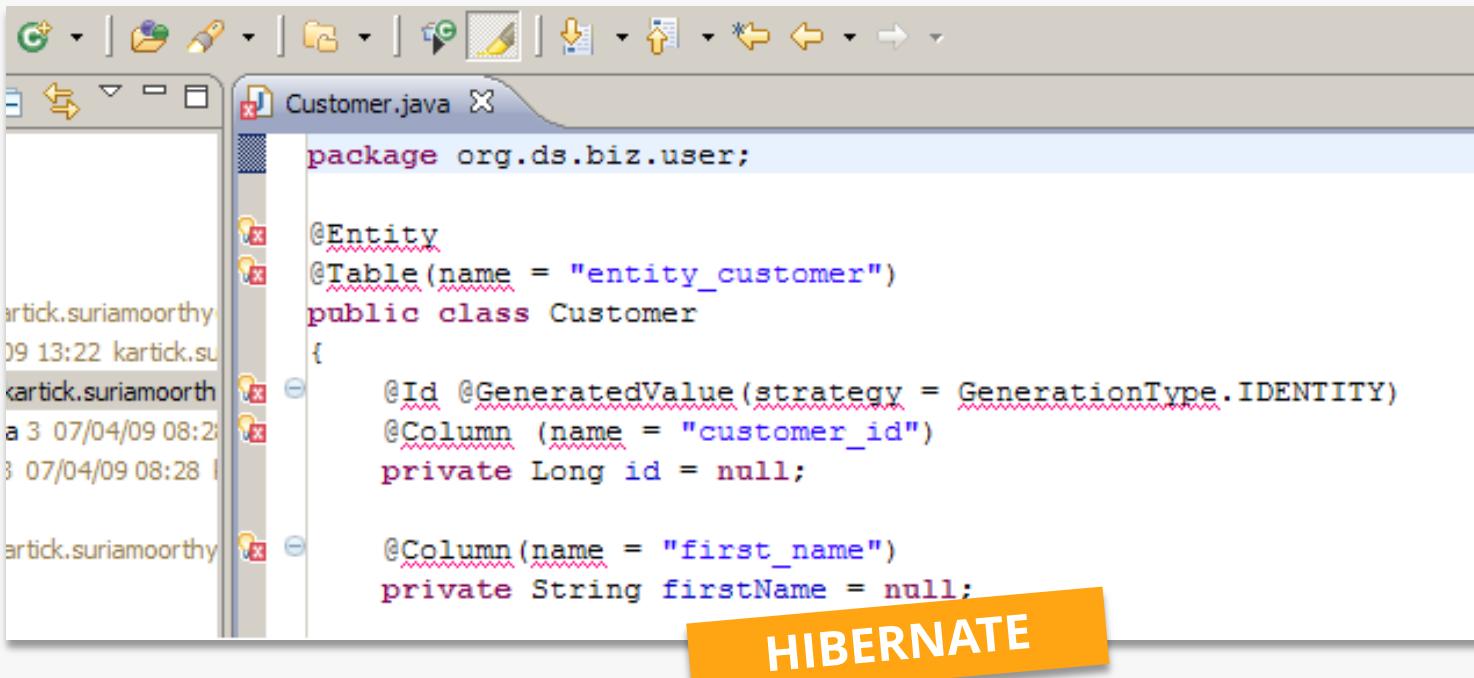
```
ClassPathScanningCandidateComponentProvider scanner = new  
ClassPathScanningCandidateComponentProvider(<DO_YOU_WANT_TO_USE_DEFALT_FILTER>);  
  
scanner.addIncludeFilter(new AnnotationTypeFilter(<TYPE_YOUR_ANNOTATION_HERE>.class));  
for (BeanDefinition bd : scanner.findCandidateComponents(<TYPE_YOUR_BASE_PACKAGE_HERE>))  
System.out.println(bd.getBeanClassName());
```



The screenshot shows a Java class browser interface. At the top, there's a navigation bar with links: Overview, Package, Class, Use, Tree, Deprecated, Index, Help, PREV CLASS, NEXT CLASS, SUMMARY: NESTED, FIELD, CONSTR, METHOD. Below the navigation bar, the class name is displayed: **org.springframework.context.annotation.ClassPathScanningCandidateComponentProvider**. It shows the class hierarchy: java.lang.Object < org.springframework.context.annotation.ClassPathScanningCandidateComponentProvider. Under "All Implemented Interfaces", it lists ResourceLoaderAware. At the bottom, it says "Direct Known Subclasses: DefinitionScanner". In the bottom right corner of the browser window, there are buttons for FRAMES, NO, DETAIL, and FIELD.

Des annotations partout

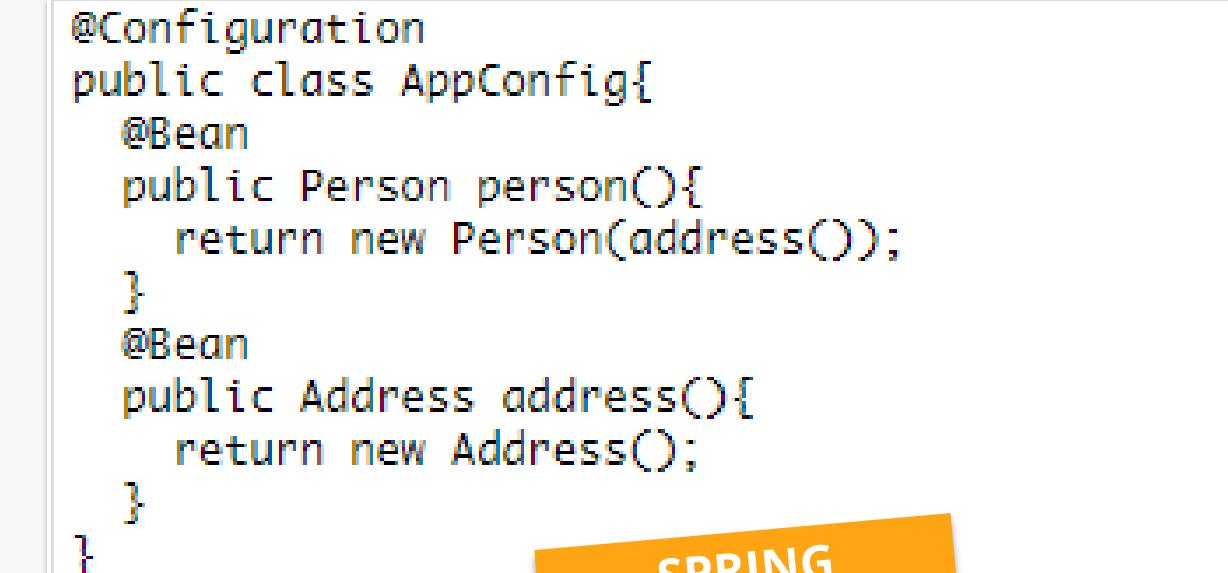
Un grand nombre de frameworks reposent massivement sur les annotations



```
Customer.java
package org.ds.biz.user;

@Entity
@Table(name = "entity_customer")
public class Customer {
    @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "customer_id")
    private Long id = null;

    @Column(name = "first_name")
    private String firstName = null;
}
```



```
@Configuration
public class AppConfig{
    @Bean
    public Person person(){
        return new Person(address());
    }
    @Bean
    public Address address(){
        return new Address();
    }
}
```



Des annotations partout

Un grand nombre de frameworks reposent massivement sur les annotations

```
@Path("/")
public class CrunchifyRESTService {
    @POST
    @Path("/crunchifyService")
    @Consumes(MediaType.APPLICATION_JSON)
    public Response crunchifyREST(InputStream incomingData) {
        StringBuilder crunchifyBuilder = new StringBuilder();
        try {
            BufferedReader in = new BufferedReader(new InputStreamReader(incomingData));
            String line = null;
            while ((line = in.readLine()) != null) {
                crunchifyBuilder.append(line);
            }
        } catch (Exception e) {
            System.out.println("Error Parsing: - ");
        }
        System.out.println("Data Received: " + crunchifyBuilder.toString());
```

JERSEY

```
2  @Getter @Setter @NoArgsConstructor // <--- THIS is it
3  public class User implements Serializable {
4
5      private @Id Long id; // will be set when persisting
6
7      private String firstName;
8      private String lastName;
9      private int age;
10
11     public User(String firstName, String lastName, int age) {
12         this.firstName = firstName;
13         this.lastName = lastName;
14         this.age = age;
15     }
16 }
```

LOMBOK

Merci

Des questions?

