

Introspection et Annotations

Manipulation
de code source Java

Introspection

Principe

- Classes et méthodes permettant
 - Accès à l'information sur les classes
 - attributs
 - méthodes
 - constructeurs
 - Manipulation des objets de ces classes
 - modification d'attributs
 - appel de constructeurs
 - appel de méthodes
- Pendant l'exécution
- Limites en Java : pas de modification des classes, ex. ajout d'attributs ou de méthodes

Utilisation *réalisation de*

- Débogueurs
- Interprètes
- Inspecteurs d'objets
- Navigateur de classes (class browsers)
- Services particuliers, ex.
 - Sérialization (sauvegarde d'objets)
 - Editeurs d'objets
 - Intercession (interception d'appels)

Principales classes

- java.lang
 - Class<T> classe
- java.lang.reflect
 - Field attribut
 - Constructor<T> constructeur
 - Method méthode

Principales classes

- **Class<T>**
 - le type de `String.class` est `Class<String>`
 - ses fields, constructors, methods, interfaces, classes, ..
- **Field**
 - son type, sa classe, sa valeur pour un objet, ..
- **Constructor<T>**
 - ses paramètres, exceptions, ..
- **Method**
 - ses paramètres, type de retour, exceptions, ..

Contextes d'utilisation

Sans volonté d'exhaustivité, quelques exemples classiques d'utilisation ...

- Inspection des méthodes
- Inspection d'objets
- Création d'objets selon des types non connus au préalable
- Appel de méthodes

Inspection des méthodes

- Éléments utilisés
 - Class, Method
 - String `getName()`
 - Class
 - static Class `forName(String c)`
 - retourne l'instance représentant la classe nommée c
 - Method[] `getMethods()`
 - retourne les méthodes publiques de la classe
 - Method
 - Class `getReturnType()`
 - Class[] `getParameterTypes()`

Inspection des méthodes

```
abstract class Produit{  
    private String reference,designation;  
    private double prixHT;  
    public Produit(){}  
    public Produit(String r,String d,double p)  
    {reference=r; designation=d; prixHT=p;}  
    public String getReference(){return reference;}  
    public void setReference(String r){reference=r;}  
    public String getDesignation(){return designation;}  
    public void setDesignation(String d){designation=d;}  
    public double getPrixHT(){return prixHT;}  
    public void setPrixHT(double p){prixHT=p;}  
    abstract public double leprixTTC();  
    public String infos(){return getReference()+" "+  
                                getDesignation()+" "+leprixTTC();}  
}
```

Inspection des méthodes

```
class ProduitNormal extends Produit
{ public ProduitNormal(){}
  public ProduitNormal(String r,String d,double p)
    {super(r,d,p);}
  public double leprixTTC()
    {return getPrixHT() * 1.196;}
}
```

```
class Livre extends ProduitNormal
{ private String editeur;
  public Livre(){}
  public Livre(String r,String d,double p,String e)
    {super(r,d,p);editeur=e;}
  public String getEditeur(){return editeur;}
  public void setEditeur(String e){editeur=e;}
  public String infos(){return super.infos()+"
                                "+getEditeur();}
}
```

Inspection des méthodes

```
package Exemples;
```

```
import java.lang.reflect.*;
```

```
// Class est dans java.lang
```

```
// Method est dans java.lang.reflect
```

Inspection des méthodes

```
public class TestReflexion
{
    public static void afficheMethodesPubliques(Class cl)
    {
        Method[] methodes = cl.getDeclaredMethods();
        for (int i=0; i<methodes.length; i++)
        {
            Method m = methodes[i];
            String m_name = m.getName();
            Class m_returnType = m.getReturnType();
            Class[] m_paramTypes = m.getParameterTypes();
            System.out.print(" "+m_returnType.getName()+
                            " "+m_name + "(");
            for (int j=0; j<m_paramTypes.length; j++)
                System.out.print(" "+m_paramTypes[j].getName());
            System.out.println(")");
        }
    }
    ..... }
```

Inspection des méthodes

```
public class TestReflexion
{ ....
public static void main(String[] argv)
    throws java.lang.ClassNotFoundException
{
    System.out.println("Saisir un nom de classe");
    Scanner s=new Scanner(System.in);
    String nomClasse = s.nextLine();
    Class c = Class.forName(nomClasse);
    TestReflexion.afficheMethodesPubliques(c);
}
} //fin TestReflexion
```

Inspection des méthodes

Saisir un nom de classe

<< Exemples.Livre

>> java.lang.String getEditeur() *Livre*

>> void setEditeur(java.lang.String)

>> java.lang.String infos()

>> double leprixTTC() *ProduitNormal*

>> double getPrixHT() *Produit*

>> java.lang.String getReference()

>> int hashCode() *Object*

>> java.lang.Class getClass()

>> boolean equals(java.lang.Object)

>> java.lang.String toString()

Inspection des objets

- Éléments utilisés
 - Object
 - Class `getClass()`
 - retourne la classe de l'objet
 - Class
 - String `getName()`
 - Field `getField(String n)`
 - retourne l'attribut nommé *n*
 - Field
 - Object `get(Object o)`
 - retourne la valeur de l'attribut pour l'objet *o*

Inspection des objets

```
Produit p = new Livre("X23", "Paroles de  
Prévert", 25, "Folio");  
System.out.println(p.getClass().getName());
```

>> Exemples.Livre

```
p = new Aliment("A21", "Pain d'épices",  
12, "BonMiel");  
System.out.println(p.getClass().getName());
```

>> Exemples.Aliment

Accès aux attributs *public*

// editeur et prixHT ont été déclarés *public* pour cette partie

```
Livre p = new Livre("X23","Paroles de  
Prévert",25,"Folio");
```

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

```
Field f1_p = p_class.getField("editeur");
```

```
Object v_f1_p = f1_p.get(p);
```

```
Field f2_p = p_class.getField("prixHT");
```

```
Object v_f2_p = f2_p.get(p);
```

```
System.out.println("v_f1_p="+v_f1_p+"  
v_f2_p="+v_f2_p);
```

>> v_f1_p=Folio v_f2_p=25.0

Accès aux attributs privés

```
f1_p.setAccessible(true);
```

- méthode héritée de
 AccessibleObject

Créer des objets

- Éléments utilisés
 - Class
 - static Class `forName(String)`
 - Constructor `getConstructor()`;
 - retourne le constructeur sans paramètres
 - Constructor
 - Object `newInstance()`
 - retourne un objet construit avec le constructeur

Créer des objets

```
System.out.println("Livre ou Aliment ?");  
Scanner s=new Scanner(System.in);  
String nomClasse = s.next();  
Object np;
```

```
// et maintenant on voudrait créer  
// un livre ou un aliment
```

Créer des objets

code classique

```
System.out.println("Livre ou Aliment ?");  
Scanner s=new Scanner(System.in);  
String nomClasse = s.nextLine();Object np;
```

```
if (nomClasse.equals("Exemples.Livre"))  
    np = new Livre();  
else if (nomClasse.equals ("Exemples.Aliment"))  
    np = new Aliment();  
else if ...
```

- **Pb extensibilité** : ajout de classe, modification de nom de classe implique :

modification de code

Créer des objets avec la réflexion

```
System.out.println("Livre ou Aliment ?");  
Scanner s=new Scanner(System.in);  
String nomClasse = s.nextLine();  
Object np;
```

```
Class c = Class.forName(nomClasse);  
Constructor constructeur=c.getConstructor();  
np = constructeur.newInstance();  
np.saisie(..); ...
```

Créer des objets avec la réflexion

- Pour appeler un constructeur prenant des paramètres

Constructor constructeur =

```
c.getConstructor(String.class,  
                String.class,  
                double.class,  
                String.class);
```

np = constructeur.newInstance

```
("xx", "Paroles", 12, "Folio");
```

Appeler des méthodes

- Éléments utilisés

- Class

- Method `getMethod(String n)`
 - retourne la méthode nommée *n*

- Method

- Object `invoke(Object)`
 - appelle la méthode sur l'objet *o*

Appeler des méthodes

```
System.out.println("Méthodes existantes sur np");  
TestReflexion.afficheMethodesPubliques(c);
```

```
>> ...  
>> java.lang.String infos()  
>> double leprixTTC()  
>> double getPrixHT()  
>> ....
```

```
System.out.println("Quelle méthode sans argument voulez-vous  
appeler ?");
```

```
String nomMeth = s.next();
```

```
<< leprixTTC
```

```
Method meth = c.getMethod(nomMeth);
```

```
Object resultat = meth.invoke(np);
```

```
System.out.println("resultat = "+resultat);
```

```
>> resultat = 14.352
```

Appeler des méthodes avec des paramètres

```
meth = c.getMethod("setEditeur",  
                    String.class);
```

```
resultat = meth.invoke(np, "Gallimard");
```

```
System.out.println("nouvel objet = "+np);
```

On peut aussi ...

- accéder aux modifieurs
- connaître les super-classes, les interfaces
- créer et manipuler des tableaux
- créer des proxys de classes ou d'instances pour intercepter des appels et ajouter du comportement (ex. tracer automatiquement)

ANNOTATIONS

Annotations

- **informations** pour les programmes traités par des outils
 - éditeurs,
 - débogueurs,
 - outils de test, documentation, statistiques, refactoring, etc.
- **tags javadoc** : annotations spécialisées pour la documentation
- d'autres formes de tags peuvent être définis par les utilisateurs (programmeurs d'outils notamment)

Utilisation dans javadoc

```
/**  
 * @deprecated As of JDK version 1.1,  
 * replaced by Calendar.get(Calendar.MONTH)  
 * as shown in {@link java.util.Calendar#get(int) get}  
 */
```

Produit dans la documentation :

getMonth

public int getMonth()

Deprecated.

***As of JDK version 1.1, replaced by
Calendar.get(Calendar.MONTH) as shown in [get](#)***

Utilisation dans javadoc

```
/**  
 * @deprecated As of JDK version 1.1,  
 * replaced by Calendar.get(Calendar.MONTH)  
 * as shown in {@link java.util.Calendar#get(int) get}  
 */
```

- block tag `@deprecated` en début de ligne
- inline tag `{@link }` en milieu de ligne
- utilisé par le programme javadoc pour créer les pages html de la documentation

Utilisation dans Eclipse

Une méthode privée inutilisée génère un warning

```
private void crypter()  
{/* à écrire plus tard*/ System.out.println("cryptage");}
```

Solution proposée par Eclipse

ajouter un tag pour faire disparaître ce warning

```
@SuppressWarnings("unused")  
private void crypter()  
{/* à écrire plus tard*/ System.out.println("cryptage");}
```

Eclipse n'affichera plus de warning !

Déclaration d'un type d'annotation

- type d' annotation = interface
- mot-clef *interface* précédé par @
- les méthodes
 - définissent des éléments
 - quand il est unique, l'élément s' appelle *value*
 - pas de paramètres
 - pas de clause throws
 - type de retour possible
 - TRP = types primitifs, String, Class, enums,
 - arrays de TRP
 - valeurs par défaut

Définition d'un type d'annotation

```
/**  
 * Request-For-Enhancement(RFE)  
 * annoter un élément à améliorer  
 * dans la version suivante  
 */  
public @interface RequestForEnhancement  
{  
    int id();  
    String synopsis();  
    String engineer() default "[unassigned]";  
    String date() default "[unimplemented]";  
}
```

Utilisation de l'annotation

~ se place comme un *modifier*

```
@RequestForEnhancement(  
    id = 23777,  
    synopsis = "Improve time complexity",  
    engineer = "Jack",  
    date = "31 oct 2009")
```

```
public static
```

```
    <T extends Comparable<? super T>>
```

```
void sort(List<T> list)
```

```
{ ... }
```

interface **Annotation**

- C'est l'interface spécialisée par les annotations
- Ne pas l'étendre manuellement

- Méthodes

Class<? extends **Annotation**> **annotationType**()
retourne le type d'annotation de cette annotation

boolean **equals**(**Object** obj)

int **hashCode**()

String toString()

Types d'Annotation de l'API

- Annotations
 - Deprecated
 - Override
 - SuppressWarnings
- Certaines portent sur d'autres annotations
 - Inherited
 - Documented
 - Retention : décrit la portée (SOURCE, CLASS, RUNTIME)
 - Target : décrit la cible (TYPE, FIELD, METHOD, etc.)

Annotation annotée

@Documented

@Retention(value=RUNTIME)

@Target(value=ANNOTATION_TYPE)

public @interface **Retention**

{ RetentionPolicy **value()**; }

Interface **AnnotatedElement**

- Implémentée par `AccessibleObject`, `Class`, `Constructor`, `Field`, `Method`, `Package`

Méthodes

`<T extends Annotation> getAnnotation`
`(Class<T> annotationType)`

retourne l'annotation du type passé en paramètre (ou null)

`Annotation[] getAnnotations ()`

retourne les annotations attachées à l'élément (incluant héritées)

`Annotation[] getDeclaredAnnotations ()`

retourne toutes les annotations attachées à l'élément (propres)

`boolean isAnnotationPresent`

`(Class<? extends Annotation> annotationType)`

retourne vrai ssi une annotation du type passé en paramètre est attachée à l'élément

Éléments pour un outil de test

- Objectif :
 - embarquer dans les classes des méthodes de test unitaire
 - annotation par les programmeurs de ces méthodes de test (pour les distinguer des autres)
 - l'outil de test utilise les annotations pour tester la classe

Type d'annotation pour les méthodes de test

```
import java.lang.annotation.*;
enum NiveauRisque {faible, moyen, eleve;}
/**
 * indique qu'une méthode est une méthode de test
 * à utiliser sur des méthodes sans paramètre
 */
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface Test
    {NiveauRisque risque();}
```

Classe en développement

```
class Foo {  
    @Test(risque=NiveauRisque.faible)  
        public static void m1()  
            {System.out.println("m1");}  
    public static void m2() {System.out.println("m2");}  
    @Test(risque=NiveauRisque.moyen)  
        public static void m3()  
            {throw new RuntimeException("Boom");}  
    public static void m4() {System.out.println("m4");}  
    @Test(risque=NiveauRisque.moyen)  
        public static void m5()  
            {System.out.println("m5");}  
    public static void m6() {System.out.println("m6");}  
    @Test(risque=NiveauRisque.eleve)  
        public static void m7()  
            {throw new RuntimeException("Crash");}  
    public static void m8() {System.out.println("m7");}  
}
```

Une classe de l'outil de test

```
import java.lang.annotation.*;
import java.lang.reflect.*;
public class TestAnnotations
{ public static void main(String[] args) throws Exception

    {int passed = 0, failed = 0;
    for (Method m :
        Class.forName(args[0]).getMethods())
    {if (m.isAnnotationPresent(Test.class) &&
        (m.getAnnotation(Test.class)).risque() != NiveauRisque.faible)
        {try {m.invoke(null); passed++;}
        catch (Throwable ex)
            {System.out.println("Test "+m+" failed:"+ex.getCause());
            failed++;}
        }}
    System.out.println("Passed: "+passed+" Failed "+failed);
}}
```

Exécution

```
>>> java TestAnnotations Foo
```

```
Test public static void Foo.m3() failed: java.lang.RuntimeException: Boom  
m5
```

```
Test public static void Foo.m7() failed: java.lang.RuntimeException: Crash  
Passed: 1 Failed 2
```

Pour aller plus loin avec les annotations

- Annotation Processing Tool (commande apt)
- s'utilise indépendamment d'une exécution
- exécute des traitements d'annotations sur un ensemble de fichiers source annotés
- Plus complexe (patron de conception Visiteur)
- Pas encore stabilisé (API mirror)
- <http://gfx.developpez.com/tutoriel/java/annotation/>
 - ramasser les Todo d'une classe
 - créer un fichier les contenant
- <http://www.javalobby.org/java/forums/t17876>