

Reflection & Annotation

Rémi Forax
forax@univ-mlv.fr

La réflexion dans les langages

- Réflexion: se connaître soi même.
 - Les 3 niveaux de réflexion :
 - **Classe des objets lors de l'exécution**,
RTTI, cast avec exception, instanceof
 - **Introspection**,
ensemble des méthodes, champs, classes internes,
appel dynamique, modification, bypass la sécurité
 - **Intercession**,
changer la façon d'effectuer l'appel de méthode,
ajouter des champs,
émuler en Java par modification du byte-code
(prog par aspect, `java.lang.instrument`)

Classe Class

- Une instance de cette classe représente une classe particulière ;
- Cette instance est unique pour une classe donnée
- L'instance permet de faire de l'introspection :
 - on peut demander pendant l'exécution de programmes les champs, les constructeurs et les méthodes de la classe, représentés par des instances de classes de **java.lang.reflect** ;
 - avec ces instances, on peut créer des objets, affecter des champs et appeler des méthodes.

Classe Class

- Pour obtenir des objets Class :
 - la construction **NomDeClasse.class** permet d'obtenir l'objet Class associé à la classe NomDeClasse.
 - la méthode **getClass()** de Object permet de connaître le type (réel) d'un objet ;
 - la méthode statique **forName** de la classe Class permet d'obtenir un objet Class à partir du nom complet de la classe ;

Class et type paramétré

- Un objet de la classe **Class** est paramétré par le type qu'elle représente

```
Class<String> clazz=String.class;  
String s=clazz.newInstance();
```

- Pour getClass(), à cause du sous-typage, la règle est **Class<? extends erasure(type déclaré)>**

```
String s="toto";  
Object o=s;  
Class<? extends Object> clazz2=o.getClass();  
  
clazz2==s.getClass() // true
```

Class et forName

- Permet de charger une classe par son nom complet (avec le nom du paquetage), pratique pour un système de plugin dynamique

```
public static void main(String[] args) throws ClassNotFoundException,
    InstantiationException, IllegalAccessException {

    Class<?> clazz=Class.forName(args[0]);
    Object obj=clazz.newInstance();

    for(Field field:clazz.getFields()) {
        System.out.println("field "+field.getName()+" "+field.get(obj));
    }
}
```

```
$ java reflect.GetFields java.awt.Point
field x 0
field y 0
```

Les types primitifs

- les types primitifs et le type **void** ont des objets **Class** correspondants.
- `type_primitif.class` est typé `Class<Wrapper>`
et pour `void`
`void.class` est typé `Class<Void>`

```
public static void main(String[] args) {  
    Class<String> clazz=String.class;  
    String s=clazz.newInstance();  
  
    Class<Integer> clazz2=int.class;  
    int i=clazz.getConstructor(Integer.class).newInstance();  
  
}
```

Propriété de la class **Class**

- Savoir si la classe est :
 - une annotation **isAnnotation()**
 - une classe anonyme **isAnonymousClass()**
 - Un tableau **isArray()**
 - Un type énuméré **isEnum()**
 - Une interface **isInterface()**
 - Une classe locale à une méthode **isLocalClass()**
 - Une classe interne/inner **isMemberClass()**
 - Une classe d'un type primitif **isPrimitive()**
 - Générée par le coompileur **isSynthetic()**

Hiérarchie de classes

- Il est possible pour une classe d'obtenir :
 - sa superclass `Class<? super T> getSuperClass()`
 - ses interfaces `Class<?>[] getInterfaces()`
- De plus, on peut :
 - Tester si un objet est de cette classe
`isInstance(Object o)`
 - Tester si une classe est sous-type d'une autre
`isAssignableFrom(Class<?> clazz)`
 - Caster une référence vers la classe
`T cast(Object o)`
 - Voir une classe comme une sous-classe
`<U> Class<? extends U> asSubclass(Class<U> clazz)`

Typesafe generics

- **cast()** et **asSubclass()** sont utilisé pour vérifier à runtime des casts non verifiable à cause de l'erasure

```
public static <T> T createNullProxy(Class<T> interfaze) {  
    Object o=Proxy.newProxyInstance(interfaze.getClassLoader(),  
        new Class<?>[]{ interfaze },EMPTY_HANDLER);  
    //return (T)o; // unsafe  
    return interfaze.cast(o); // ok safe  
}
```

```
public static Class<? extends LookAndFeel> getLafClass(String className) {  
    Class<?> lafClass=Class.forName(className);  
    //return (Class<? extends LookAndFeel>)o; // unsafe  
    return lafClass.asSubClass(LookAndFeel.class); // ok safe  
}
```

Créer un objet à partir de sa Class

- La classe `Class<T>` possède une méthode **`newInstance()`** qui renvoie un objet de type `T`

```
public static void main(String[] args) throws ... {  
    Class<String> clazz=String.class;  
    String s=clazz.newInstance();  
}
```

- Lève les exceptions suivantes :
 - **`InstantiationException`**, si la classe est abstraite ou qu'il n'existe pas de constructeur sans paramètre
 - **`IllegalAccessException`**, si le constructeur n'est pas publique
 - **`InvocationTargetException`**, si une exception est levée par le constructeur, celle-ci est stockée dans la cause de l'exception

java.lang.reflect

- A partir d'une class on peut accéder à l'ensemble de ces membres :
 - les constructeurs d'objets de type T sont représentés par des instances de la classe **Constructor<T>**.
 - Les champs sont représentés par des instances de la classe **Field** ;
 - les méthodes sont représentées par des instances de la classe **Method** ;
 - Les classes internes sont représentées par des instances de la classe **Class** ;

java.lang.reflect

- Avec `XXX`, `Constructor`, `Field`, `Method` ou `Class` :
 - la méthode `getDeclaredXXXs()`, qui renvoie tous les `XXX` (privés inclus) qui sont déclarés par la classe, c'est-à-dire non hérités ;
 - la méthode `getXXXs()`, qui retourne tous les `XXX` **publics**, y-compris ceux qui sont hérités.
 - la méthode `getDeclaredXXX(param)`, qui renvoie le `XXX` **déclaré**, dont le nom et/ou le type des paramètres est donné en argument ;
 - la méthode `getXXX(param)`, qui retourne le `XXX` **public**, hérité ou non, dont le nom et/ou les types (objets `Class`) des paramètres sont donnés en argument ;

java.lang.reflect

- Les instances des classes `Class`, `Constructor`, `Method` ou `Field` ne sont **pas liées à un objet** particulier.
- Lorsque l'on veut changer la valeur d'un champs d'un objet ou appelée une méthode sur celui-ci, on doit **fournir l'objet** en paramètre
- Pour les méthodes ou les champs **statiques**, ce paramètre peut-être **null**.

Classe Constructor

- On obtient un `Constructor<T>` à partir de la classe en utilisant **`getConstructor(Class<?>... types)`**
- La méthode **`T newInstance(Object.. args)`** permet d'appeler le constructeur avec des arguments

```
public static void main(String[] args) throws NoSuchMethodException,
    InstantiationException, IllegalAccessException, InvocationTargetException {

    Class<Point> point = Point.class;
    Constructor<Point> c = point.getConstructor(int.class, int.class);
    Point p = c.newInstance(1,2);
    System.out.println(p); // (1,2)
}
```

Classe Field

- Une instance de **Field** représente un champs d'un objet
 - Object **get***(Object target) permet d'obtenir la valeur
 - void **set***(Object target) permet de changer la valeur

```
public static void main(String[] args)
    throws NoSuchFieldException, IllegalAccessException {

    Field xField=Point.class.getField("x");
    Point p=new Point(1,1);
    System.out.println(xField.getInt(p)); // 1
    xField.setInt(p,12);
    System.out.println(p); // (12,1);
}
```

Classe Method

- Une instance de **Method** obtenu avec **getDeclaredMethod(String, Class...)** permet d'appeler une méthode sur tous les objets d'un même type (et sous-types) ;
- La méthode **Object invoke(Object.. args)** permet d'appeler la méthode avec des arguments

```
public static void main(String[] args) throws NoSuchMethodException,
    IllegalAccessException, InvocationTargetException {

    Class<PrintStream> print=PrintStream.class;
    Method method = print.getDeclaredMethod("println", char.class);
    method.invoke(System.out, 'a'); // a
}
```

Inner class & constructeur

- Pour **inner-class**, il faut penser à ajouter un argument aux constructeurs correspondant à une instance de la classe englobante.

```
public class InnerTest {
    class Inner { // pas public ici
        public Inner(int value) {
            this.value=value;
        }
        @Override public String toString() {
            return String.valueOf(value);
        }
        private final int value;
    }
    public static void main(String[] args) throws InstantiationException,
        IllegalAccessException, InvocationTargetException, NoSuchMethodException {
        Class<?> innerClass=InnerTest.class.getDeclaredClasses()[0];
        Object o=innerClass.getConstructor(InnerTest.class,int.class).
            newInstance(new InnerTest(),3);
        System.out.println(o);
    } }
}
```

Visibilité des éléments

- La classe `java.lang.reflect.Modifier` permet d'interpréter un entier renvoyé par les éléments comme modificateurs de visibilité

```
for(Field field:String.class.getDeclaredFields()) {
    int modifiers=field.getModifiers();
    StringBuilder builder=new StringBuilder();
    if (Modifier.isPrivate(modifiers))
        builder.append("private ");
    if (Modifier.isProtected(modifiers))
        builder.append("protected ");
    if (Modifier.isPublic(modifiers))
        builder.append("public ");
    if (Modifier.isStatic(modifiers))
        builder.append("static ");
    if (Modifier.isFinal(modifiers))
        builder.append("final ");
    System.out.println(
        builder.append(field.getType().getName()).append(' ').
        append(field.getName()));
}
```

java.lang.reflect et Sécurité

- Par défaut, la réflexion vérifie la sécurité lors de l'exécution, on ne peut alors effectués les opérations que si l'on a les droits

```
public static void main(String[] args) throws NoSuchMethodException,
    InvocationTargetException, InstantiationException, IllegalAccessException {
    // access to package private constructor which shares value array
    // beware it's SUN specific !!
    Constructor<String> c=String.class.getDeclaredConstructor(
        int.class,int.class,char[].class);

    char[] array="hello".toCharArray();
    String name=c.newInstance(0,array.length,array);
}
```

```
Exception in thread "main" java.lang.IllegalAccessException: Class
reflect.UnsafeTest can not access a member of class java.lang.String with
modifiers ""
    at sun.reflect.Reflection.ensureMemberAccess(Reflection.java:65)
    at java.lang.reflect.Constructor.newInstance(Constructor.java:505)
    at reflect.UnsafeTest.main(UnsafeTest.java:16)
```

Passer outre la Sécurité

- Les Constructor, Field et Méthod hérite de AccessibleObject qui possède un méthode setAccessible qui permet d'éviter de faire le test de sécurité
 - de passer outre la sécurité
 - d'accélérer le code
- On peut alors appelé des méthodes privées, changer la valeur d'un champs final (pas en 1.3 et 1.4 ??), etc.

Exemple de setAccessible()

- On utilise **setAccessible()** sur la méthode pour dire de passer outre la sécurité

```
public static void main(String[] args) throws NoSuchMethodException,
    InvocationTargetException, InstantiationException, IllegalAccessException {

    // access to package private constructor which shares value array
    // beware it's SUN specific !!
    Constructor<String> c=String.class.getDeclaredConstructor(
        int.class,int.class,char[].class);
    c.setAccessible(true);

    char[] array="hello".toCharArray();
    String name=c.newInstance(0,array.length,array);

    System.out.println(name); // hello
    array[4]='\n';
    System.out.println(name); // hell
}
```

Les tableaux par reflexion

- `java.lang.reflect.Array` permet de créer ou de manipuler des tableaux de type primitif ou d'objet
- Création d'un tableau
 - static Object **newInstance**(Class<?> componentType, int length)
- Obtenir la valeur d'une case
 - static Object **get***(Object array, int index)
- Changer la valeur d'une case
 - static void **set***(Object array, int index, Object value)

Exemple du robot

```
public class Robot {
    public enum Direction {
        NORTH, EAST, SOUTH, WEST;
    }
    private final static Direction[] DIRS=Direction.values();
    private Direction dir=Direction.NORTH;
    private void turn(int pos) {
        int val=dir.ordinal()+pos;
        val=((val<0)?val+DIRS.length:val)%DIRS.length;
        dir=DIRS[val];
    }
    public void turnLeft() {
        turn(-1);
    }
    public void turnRight() {
        turn(1);
    }
    public void turnAround() {
        turn(2);
    }
    public void turnDir() {
        System.out.println(dir);
    }
}
```

dir
NORTH
right
left
dir
NORTH
left
dir
EAST
around
dir
WEST
dir
WEST

Exemple du robot

- Il y a un bug dans ce code (cf plus tard)

```
public class Robot {
    ...
    public static void main(String[] args) throws IllegalArgumentException,
        IllegalAccessException, InvocationTargetException {

        HashMap<String,Method> map=new HashMap<String,Method>();
        for(Method m:Robot.class.getMethods())
            map.put(m.getName().substring(4).toLowerCase(),m);

        Robot robot=new Robot();
        Scanner scanner=new Scanner(System.in);
        while (scanner.hasNextLine()) {
            String action=scanner.nextLine();
            Method method = map.get(action.toLowerCase());
            if (method!=null)
                method.invoke(robot);
        }
    }
}
```

Reflection et Type paramétrée

- Par défaut, les méthodes de reflection retournes les types érasées par le compilateur et non les types paramétrés définies par l'utilisateur
- Il est possible d'obtenir :
 - Les variable de types des types et méthodes paramétrées
 - La signature exacte (avec les generics) des méthodes

Reflection et Type paramétrée

- Par défaut, les méthodes de reflection retournes les types érasées par le compilateur et non les types paramétrés définies par l'utilisateur
- L'ensemble des méthodes de reflection possède donc deux versions :
 - une version érasé utilisant la classe **Class** pour représenté les types, ex: `getExceptionTypes()`
 - une version paramétré préfixé par `getGenerics...` utilisant l'interface **Type**, ex: `getGenericsExceptionTypes()`

L'interface Type

- L'interface **Type** définit l'interface de base de tous les types Java :
 - Les types paramétrés, **ParameterizedType**
 - Les variables de type (T), **TypeVariable**<D>, D correspond au type sur lequel il est déclaré
 - Les wildcards, **WildcardType**,
 - Les classes, **Class**
 - Les tableaux, **GenericArrayType**

ParameterizedType

- Représente les types paramétrés
ex: `List<Integer>`
- Les méthodes :
 - Les types arguments (ici, `Integer`)
 - `Type[] getActualTypeArguments()`
 - Le type *raw* (ici, `List`)
 - `Type getRawType()`
 - Le type englobant (pour les inner-classes) ou `null`
 - `Type getOwnerType()`

TypeVariable

- **TypeVariable**<D> représente une variable de type déclarée par D.
 - Le nom de la variable (ex: T)
 - String getName()
 - Ses bornes (ex: Object)
 - Type[] getBounds()
 - L'élément déclarant (Class, Method ou Constructor)
 - D getGenericDeclaration()
- L'interface **GenericDeclaration** représente un élément pouvant déclarée une variable de type :
 - TypeVariable<?>[] getTypeParameters().

WildcardType & GenericArrayType

- **WildcardType** : un wildcard avec soit une borne supérieur soit une borne inférieur
 - Les bornes inférieurs
 - `Type[] getLowerBounds()`
 - Les bornes supérieurs
 - `Type[] getUpperBounds()`
- **GenericArrayType** : un tableau
 - Le type contenu du tableau
`Type getGenericComponentType()`

Exemple

```
public static void main(String[] args) {
    System.out.println("List "+
        Arrays.toString(List.class.getTypeParameters()));

    for(Method m:List.class.getMethods()) {
        System.out.println(m.getGenericReturnType()+" "+m.getName()+" "+
            Arrays.toString(m.getGenericParameterTypes()));
    }
}
```

```
List [E]
int hashCode []
void add [int, E]
boolean add [E]
int indexOf [class java.lang.Object]
void clear []
boolean equals [class java.lang.Object]
boolean contains [class java.lang.Object]
boolean isEmpty []
int lastIndexOf [class java.lang.Object]
boolean addAll [int, java.util.Collection<? extends E>]
...
```

Classe Proxy

- La classe Proxy permet de construire dynamiquement une instance qui implémente un ensemble d'interface
 - Object **newProxyInstance**(ClassLoader loader, Class<?>[] interfaces, InvocationHandler h)
- L'interface **InvocationHandler** possède une méthode **invoke** qui sera appelée pour chaque méthode des interfaces et de Object.
 - Object invoke(Object proxy, Method method, Object[] args)

Exemple de Proxy

- Renvoie une liste qui affiche les opérations effectués

```
@SuppressWarnings("unchecked")
public static <T> List<T> traceList(final List<T> list) {
    return (List<T>)Proxy.newProxyInstance(
        ProxyTest.class.getClassLoader(),
        new Class<?>[]{List.class},
        new InvocationHandler() {
            public Object invoke(Object proxy, Method method, Object[] args)
                throws Throwable {

                System.out.println("enter "+method);
                try {
                    return method.invoke(list, args);
                } finally {
                    System.out.println("exit "+method);
                }
            }
        }
    );
}
```

Annotation

- Les annotations sont des indications présentes dans le code associées à différents éléments du code
 - Java :
 - permet d'annoter les types, méthodes, champs etc. avec des annotations
 - possède des annotations déclarées dans le JDK (@Override, @SuppressWarnings, etc)
 - permet de déclarer ses propres annotations ou utiliser celle déclarées d'autres bibliothèques

Annotation

- Pour Java, une annotation est :
 - lors de sa déclaration, une interface
 - pour son utilisation, une classe implémentant l'interface

```
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface Action {
    String value() default "";
}
```

déclaration

Utilisation

```
class Robot {
    @Action("left")
    public void turnLeft() {
        turn(-1);
    }
}
```

Déclarer une annotation

- Une annotation se déclare comme une interface, mais avec **@interface**.
- Chaque paramètre de l'annotation est une méthode sans argument, dont le type de retour doit être valide.
- On peut fournir un argument par défaut avec le mot-clef **default**.
- La méthode par défaut est **value()**.

Utiliser une Annotation

- Il est possible d'utiliser une annotation sur les paquetages, les types, les méthodes, les constructeurs, les paramètres de méthodes, les variable local

```
@Override @Override  
void m() {...}
```

← Ne marche pas

- Une seule annotation par élément
- On ne peut pas utiliser d'annotations sur des paramètre de type (attendre jdk7: JSR 308)

Valeurs d'une Annotation

- La valeur d'un champ d'une annotation doit être une valeur non null **constante** pour le compilateur
 - Les types suivants sont acceptés :
 - les types primitifs
 - les énumérations
 - le type **Class**
 - les chaînes de caractères (**String**)
 - des annotations, sans circularité
 - des tableaux des types ci-dessus

Exemples d'Annotations

- Annotation avec une seule valeur

```
public @interface Test {  
    String value() default "";  
}
```

```
@Test int i;  
@Test("test2")  
void m() {...}
```

- Annotation avec plusieurs valeurs

```
public @interface Test2 {  
    Class<?> type();  
    int number() default 2;  
}
```

```
@Test2(type=String.class, number=3)  
class Z {...}
```

```
void m(@Test2(type=int.class) int val)
```

- Annotation avec un tableaux

```
public @interface Test3 {  
    int[] value();  
}
```

```
@Test3({2, 4, 5})  
enum W {...}
```

Exemples d'Annotations (suite)

- Annotation avec une autre annotation

```
public @interface Version {  
    String value();  
    Author[] authors();  
}  
public @interface Author {  
    String value();  
}
```

```
@Version(value="1.12.36",  
         authors={  
             @Author("remi"),  
             @Author("julien")  
         })  
class G {...}
```

- Annotation sur un paquetage, doit être mis dans le fichier **package-info.java**

```
@Version("1.12.36")  
package fr.umliv.mypackage;
```

Les Meta-annotations

- Les annotations sont elles-même annotées par des annotations pour indiquer :
 - **@Target** : sur quels elements peut-on mettre une annotation (type, méthode variable local etc.)
 - **@Retention** : l'annotation n'est visible que pour le compilateur, le chargeur de classe ou par reflexion
 - **@Inherit** : Pour une annotation déclarée sur un type, l'annotation est visibles dans les sous-types
 - **@Documented** : visible dans javadoc

Accéder aux annotations

- En fonction de la méta-annotations `@Retention` les annotations sont disponibles :
 - `RetentionPolicy.SOURCE`, l'annotation n'est vu que par le compilateur ou par un parser d'annotation
 - `RetentionPolicy.CLASS`, l'annotation est vu par le compilateur et par le chargeur de classe
 - `RetentionPolicy.RUNTIME`, l'annotation est vu par le compilateur, par le chargeur de classe et par réflexion
- On récupère les annotations à runtime par réflexion, à partir de l'élément correspondant (Class, Field, Method, etc).

Reflection et annotation

- Les éléments annotables implante **AnnotatedElement**
 - Une annotation d'un certain type est présente:
 - boolean **isAnnotationPresent**(Class<? extends Annotation> annotationClass)
 - Demande une annotation spécifique (ou null)
 - <T extends Annotation> T **getAnnotation**(Class<T> annotationClass)
 - Demande les annotations déclarées
 - Annotation[] **getDeclaredAnnotations**()
 - Demande toutes les annotations (@Inherit)
 - Annotation[] **getAnnotations**()

Annotation plus exotiques

- Les annotations sur les paramètres sont récupérables à partir d'une méthode ou d'un constructeur
 - `Annotation[] (Method|Constructor)getParameterAnnotations()`
- Il n'est pas possible de récupérer les annotations sur les variables locales à l'exécution (JSR 308 ?)

Exemple du robot (suite)

```
public class Robot {
    public enum Direction {
        NORTH, EAST, SOUTH, WEST;
    }
    private final static Direction[] DIRS=Direction.values();
    private Direction dir=Direction.NORTH;
    private void turn(int pos) {
        int val=dir.ordinal()+pos;
        val=((val<0)?val+DIRS.length:val)%DIRS.length;
        dir=DIRS[val];
    }
    @Action("left") public void turnLeft() {
        turn(-1);
    }
    @Action("right") public void turnRight() {
        turn(1);
    }
    @Action("around") public void turnAround() {
        turn(2);
    }
    @Action() public void dir() {
        System.out.println(dir);
    }
}

@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface Action {
    String value() default "";
}
```

Exemple du robot (fin)

- On utilise le nom de la méthode en cas de ""

```
public class Robot {
    ...
    public static void main(String[] args)
        throws IllegalAccessException, InvocationTargetException {

        HashMap<String,Method> map=new HashMap<String,Method>();
        for(Method m:Robot.class.getMethods()) {
            Action action=m.getAnnotation(Action.class);
            if (action==null)
                continue;
            String name=action.value();
            if (name.isEmpty())
                name=m.getName();
            map.put(name.toLowerCase(),m);
        }
        ..
    }
}
```

Annotation à compile time

- Il existe deux outils de traitement à compile-time des annotations :
 - **apt** (annotation processor tool) est un outils de traitement des annotations (déprécié en 1.6)
 - **javac** (1.6)
 - **javax.annotation.processing** définit les processeurs d'annotation
 - **javax.lang.model** définit des objets pour chaque partie du code
 - On utilise l'option `-processor` pour spécifier le ou les processeur(s) d'annotations

```
javac -processor reflect.MyProcessor source-files
```

Exemple de processor

- Affiche les fichiers pris en paramètre ainsi que les annotations de ceux-ci

```
import javax.annotation.processing.AbstractProcessor;
import javax.annotation.processing.RoundEnvironment;
import javax.annotation.processing.SupportedAnnotationTypes;
import javax.annotation.processing.SupportedSourceVersion;
import javax.lang.model.SourceVersion;
import javax.lang.model.element.TypeElement;

@SupportedAnnotationTypes("*")
@SupportedSourceVersion(SourceVersion.RELEASE_6)
public class MyProcessor extends AbstractProcessor {
    @Override
    public boolean process(Set<? extends TypeElement> annotations,
        RoundEnvironment env) {

        System.out.println(env.getRootElements());
        System.out.println(annotations);
        return false;
    }
}
```

java.lang.ClassLoader

- Object utilisé pour charger dynamiquement les classes Java lorsque celle-ci sont demandées
 - Par défaut, il existe deux classes loaders :
 - Le classloader primordial des classes du JDK accessibles par le bootclasspath, ce classloader n'est pas accessible
 - Le classloader système qui charge les classes de l'application
`ClassLoader.getSystemClassLoader()`
- On peut de plus créer son propre classloader

Charger une classe

- Le classloader possède une méthode **loadClass()** qui permet de charger une classe par son nom (avec le paquetage) :
- si la classe n'est pas trouvée une exception **ClassNotFoundException** est levée

```
public static void main(String[] args)
    throws ClassNotFoundException {

    ClassLoader loader=ClassLoader.getSystemClassLoader();
    System.out.println(loader.loadClass("java.lang.Integer"));
}
```

- La classe est chargée mais pas forcément initialisée

Charger une classe (suite)

- Il y a deux autres façons raccourcies qui utilisent le classloader de la classe courante :
 - Utiliser **Class.forName()**, qui lève une exception **ClassNotFoundException**
 - Utiliser la notation **.class**, qui lève une erreur **ClassNotFoundException** est levée

```
public class Test {  
    public static void main(String[] args)  
        throws ClassNotFoundException {  
  
        System.out.println(Integer.class);  
        System.out.println(Class.forName("java.lang.Integer"));  
        System.out.println(  
            Test.class.getClassloader().load("java.lang.Integer"));  
    }  
}
```

Class et ClassLoader

- Chaque classe connaît son classloader que l'on peut obtenir avec la méthode `clazz.getClassLoader()`
- Si la classe est chargée par le classloader primordiale, `getClassLoader()` renvoie **null**

```
public static void main(String[] args) {  
    System.out.println(String.class.getClassLoader());  
    // null  
    System.out.println(ClassLoader.getSystemClassLoader());  
    //sun.misc.Launcher$AppClassLoader@fabe9  
    System.out.println(ClassLoaderTest.class.getClassLoader());  
    //sun.misc.Launcher$AppClassLoader@fabe9  
}
```

ClassLoader et cache

- Le mécanisme des classloaders garantie que si une même classe est chargée deux fois, celui-ci renverra la même instance.
- Pour cela, un classloader possède un cache de toutes les classes qu'il a déjà chargées

```
public static void main(String[] args)
    throws ClassNotFoundException {

    System.out.println(
        String.class==Class.forName("java.lang.String")); // true
}
```

URLClassLoader

- Implantation d'un class loader qui peut aller chercher des classes en indiquant plusieurs URLs désignant
 - soit des répertoires
 - soit des jars

```
public static void main(String[] args
    throws MalformedURLException, ClassNotFoundException {

    URLClassLoader loader = new URLClassLoader(new URL[]{
        new URL("http://monge.univ-mlv.fr/~forax/tmp/")});

    Class<?> printerClass = loader.loadClass ("reflect.PrinterImpl");
    System.out.println(printerClass); // class reflect.PrinterImpl
}
```

ClassLoader parent

- Un classloader possède un parent (**getParent()**)
- Le mécanisme des classloaders oblige (normalement) un classloader à demander à son parent s'il ne peut pas charger une classe **avant** d'essayer de la charger lui-même.

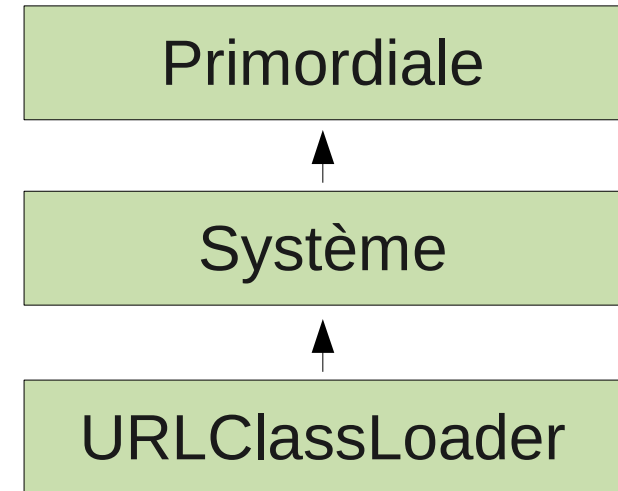
```
public static void main(String[] args
    throws MalformedURLException, ClassNotFoundException {

    URLClassLoader loader = new URLClassLoader(new URL[]{
        new URL("http://monge.univ-mlv.fr/~forax/tmp/")});

    - String class est chargée par le classloader primordiale
    Class<?> stringClass = loader.loadClass ("java.lang.String");
    System.out.println(stringClass==String.class); // true
}
```

ClassLoader et délégation

- Ce mécanisme de délégation permet à un classloader de connaître les classes systèmes en déléguant leurs chargements au classloader primordiale



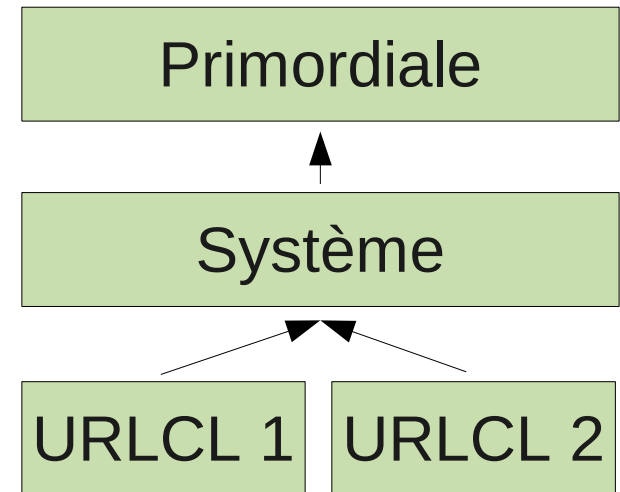
```
public static void main(String[] args
    throws MalformedURLException, ClassNotFoundException {

    URLClassLoader loader = new URLClassLoader(new URL[]{
        new URL("http://monge.univ-mlv.fr/~forax/tmp/")});

    Class<?> stringClass = loader.loadClass ("java.lang.String");
    System.out.println(stringClass==String.class); // true
}
```

Plusieurs class loaders

- Une classe reste attachée au classloader qui l'a chargée donc deux classes chargées par des classloaders différents sont différentes



```
URL url=new URL("http://monge.univ-mlv.fr/~forax/tmp/");
URLClassLoader loader = new URLClassLoader(new URL[]{url});
```

```
Class<?> c1 = loader.loadClass ("reflect.PrinterImpl");
Object o1 = c1.newInstance();
```

```
URLClassLoader loader2 = new URLClassLoader(new URL[]{url});
Class<?> c2 = loader2.loadClass ("reflect.PrinterImpl");
Object o2 = c2.newInstance();
```

```
System.out.println(c1==c2); // false
```

```
System.out.println(o1.equals(o2)); // false à cause du instanceof
```

ClassLoader API

- Les méthodes de chargement des classes :
 - Charge la classe en demandant au père d'abord
 - **loadClass**(String name)
 - Charge une classe localement
 - **findClass**(String name)
 - Insère une classe dans la VM par son bytecode
 - **defineClass**(String name, byte[] b, int off, int len)
- Si l'on veut respecter le mécanisme de délégation on redéfinie **findClass** et pas **loadClass**.

ClassLoader API (suite)

- Les méthodes de chargement de ressources :
- Trouver une ressources associé à un classloader
 - URL **getResource**(String name)
 - InputStream **getResourceAsStream**(String name)
 - Enumeration<URL> **getResources**(String name)
- Trouver une ressources système :
 - static URL **getSystemResource**(String name)
 - static InputStream **getSystemResourceAsStream**(String name)
 - static Enumeration<URL> **getSystemResources**(String name)

Décharger une classe

- Le fait que tout les objets d'une classe ne soient pas atteignable n'est pas suffisant pour décharger une classe
- Chaque classloader possède un cache de l'ensemble des classes chargées donc il faut que le classloader ayant chargée les classes soit collecté pour décharger les classes chargées

Décharger une classe

```
URLClassLoader loader = new URLClassLoader(new URL[]{
    new URL("http://monge.univ-mlv.fr/~forax/tmp/")});

Class<? extends Printer> printerClass=
    loader.loadClass("reflect.PrinterImpl").asSubclass(Printer.class);
Printer printer= printerClass.newInstance();
printer=null;
System.gc();
printerClass=null; // la classe reste stocké dans le classloader
System.gc();
loader=null;
System.gc(); // le classloader est collecté, les classes sont déchargées
```

```
$java -verbose:class UnloadClass
...
[Loaded reflect.Printer from file:/C:/java/workspace/java-
avancé/classes/]
[Loaded reflect.PrinterImpl from http://monge.univ-
mlv.fr/~forax/tmp/]
[Unloading class reflect.PrinterImpl]
```