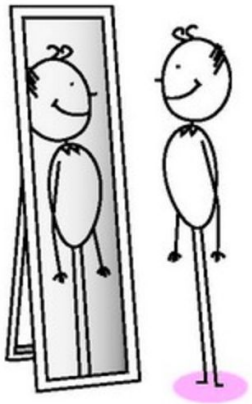
# INTROSPECTION

# Introspection / Réflexivité

➢ **Un langage est dit réflexif s'il offre des constructions permettant d'écrire un programme qui examine sa propre structure.**



➢ **Consulter les caractéristiques d'un objet et de sa classe**

- champs, fonctions, constructeurs, super classe, interfaces implémentées

➢ **Modifier les valeurs des champs**

➢ **Appeler les fonctions et les constructeurs, sans connaître leurs noms**

# En Java

- **Classe java.lang.Class**

- **API java.lang.reflect**
  - Classes Method Constructor Field Package Modifier ….

- **Utilisé largement dans les environnements Java**
  - Machine virtuelle, processus de sérialisation, complétion de code dans les environnements de développement, écriture de code générique, générateurs de code, outils de log, debugger, frameworks, …
  - Favorise l'extensibilité d'une application par le biais de plug-in

# A ne pas confondre avec la commande javap

- Permet de décompiler un fichier .class
- Exemple : javap geometrie.Point
- Options
  - -c : bytecode des instructions
  - -s : codification des noms de classes/types
  - -verbose : mode (très) bavard

Consulter rapidement une classe dans un terminal

Restreint à la consultation
Beaucoup d'informations sous forme de texte

# Notion de ClassLoader

- Objet de la JVM qui charge dynamiquement les classes
- Trois, par défaut
  - ClassLoader de bootstrap, implémenté en code natif : charge les classes de base, dont la classe `ClassLoader`
  - ClassLoader d'extension
  - ClassLoader d'application : charge les classes définies dans le classpath
- Créer un ou plusieurs ClassLoader personnalisés
  - Séparer plusieurs applications dans une même JVM
  - Charger / Recharger de nouvelles classes
  - Changer de ClassLoader (*cf.* Projet S4)

# La classe java.lang.Class

- Toute classe chargée par un ClassLoader est décrite par un objet, instance de la classe Class.

```
String ch = ”Un exemple” ;
Class<?> cch = ch.getClass() ;
Class<?> cp = Point.class ;
```

- Tout type est décrit par une instance de Class.

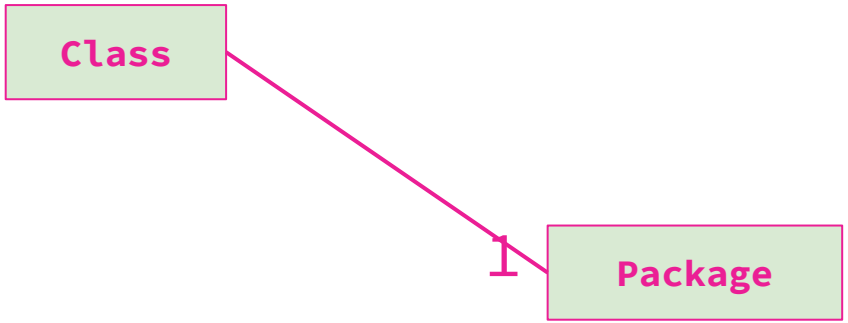
```
Class<?> cch = String[].class ;
Class<?> ci = int.class ;
```

# La classe java.lang.Class …

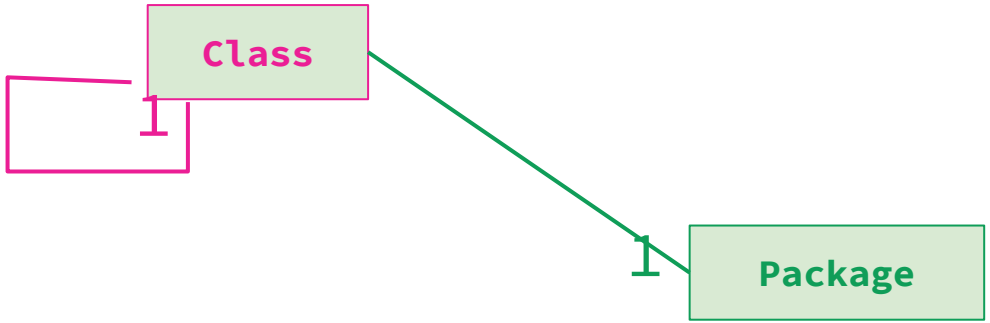
- L'introspection se fait à partir des instances de la classe `Class`.
- Peu d'intérêt sur des classes connues (comme `String` ou `Point`)

- Demander le chargement de nouvelles classes (extension dynamique)

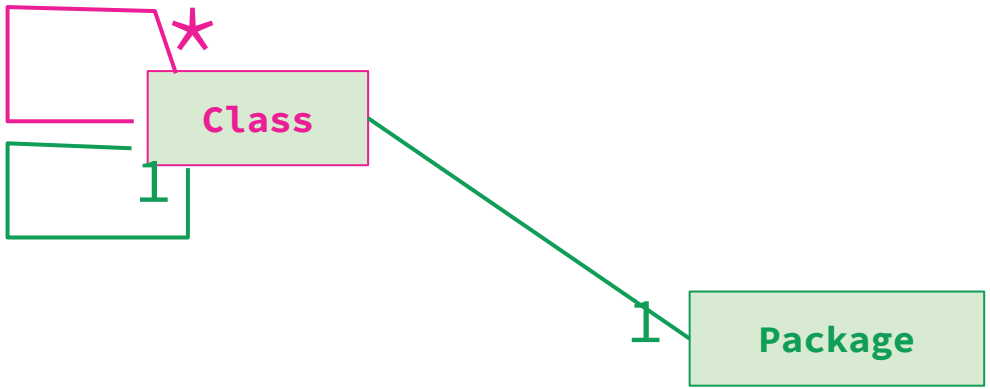
```
Scanner scan = new Scanner(System.in) ;
String nom = scan.nextLine() ;
Class<?> cn = Class.forName(nom) ;
```



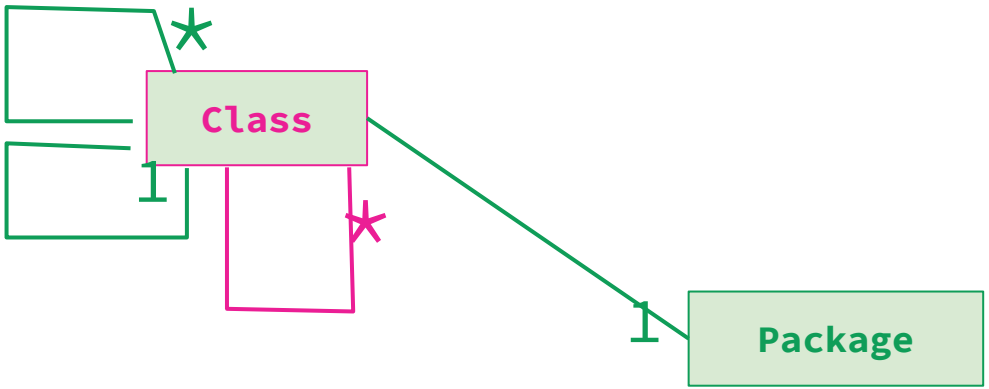

Chaque classe est rattachée à un package.



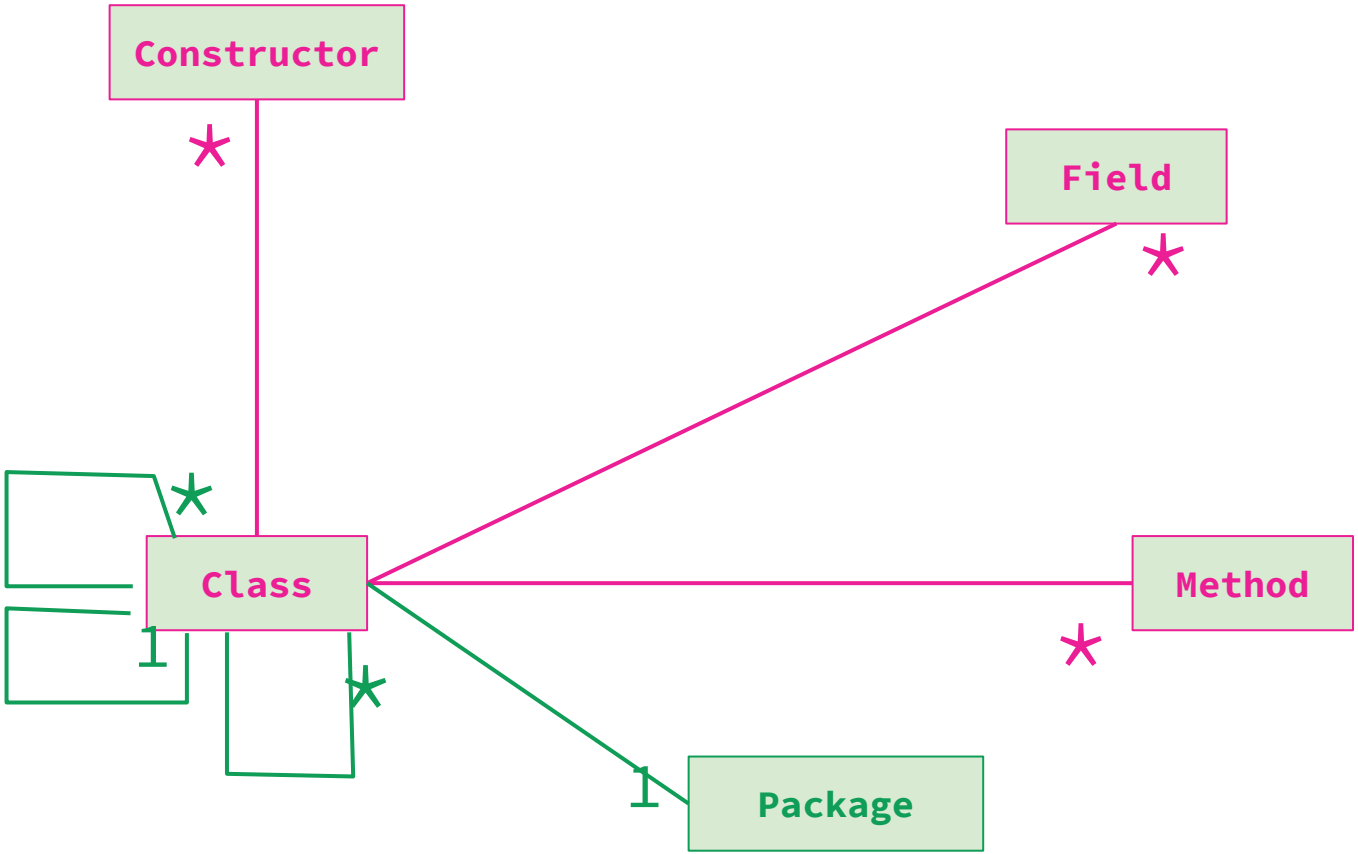

Chaque classe a une super classe.



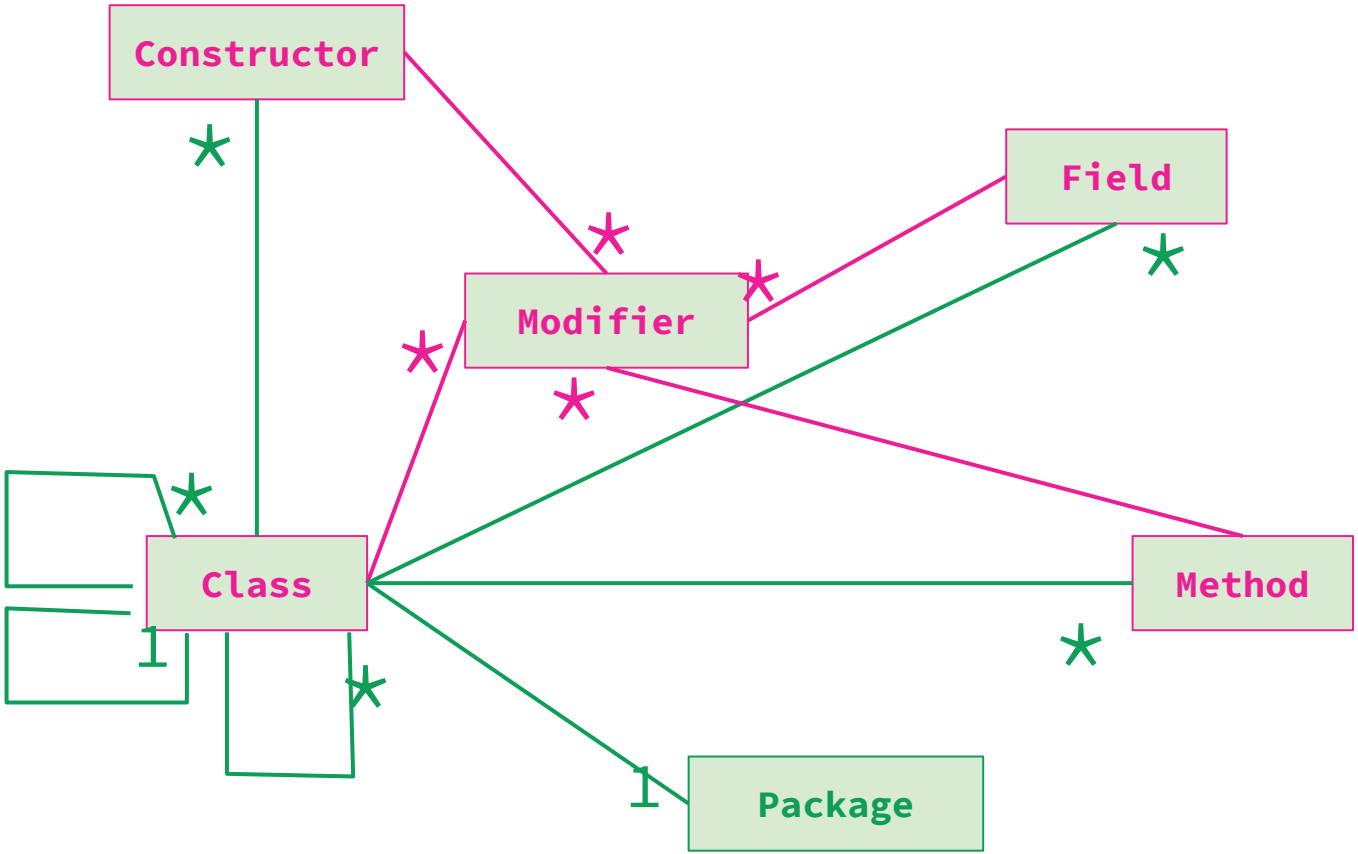

Chaque classe peut implémenter plusieurs interfaces.




Chaque classe peut contenir la définition d'autres classes.

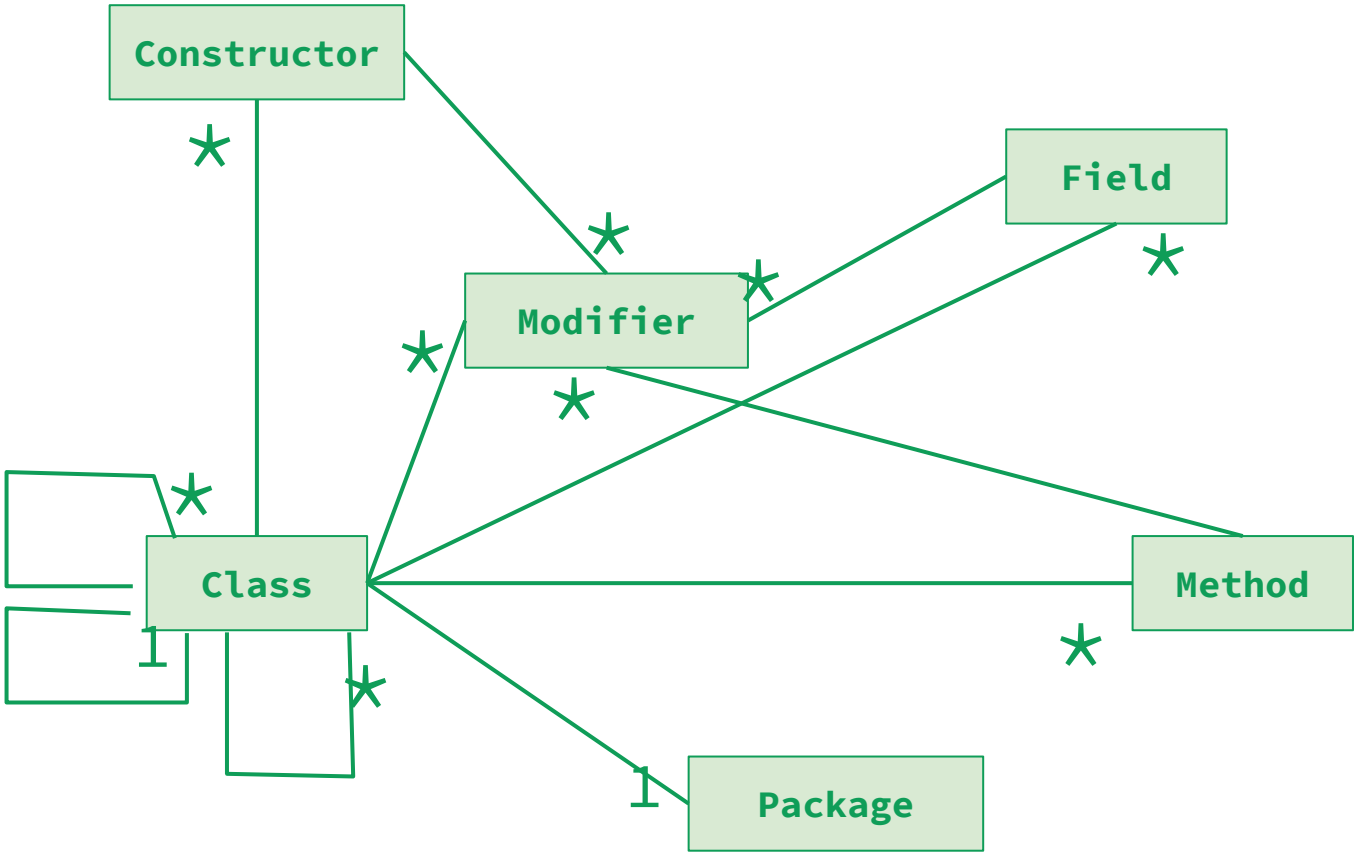


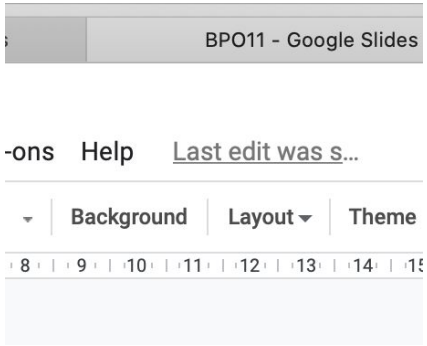

Chaque classe définit des constructeurs, champs et méthodes.

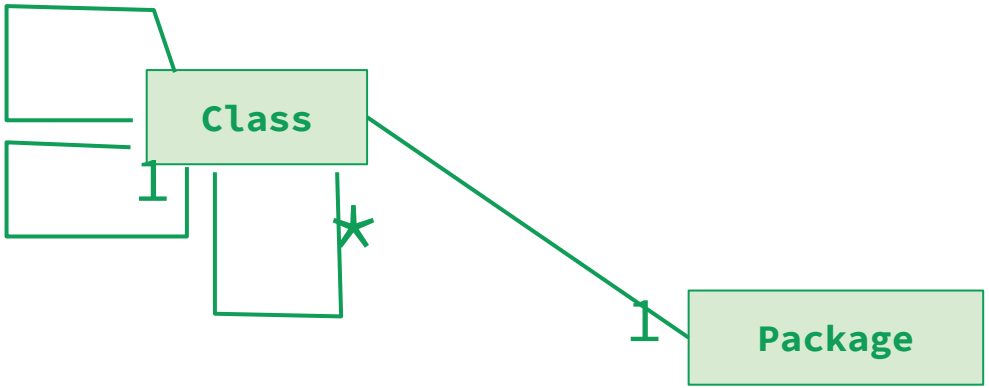



Chaque élément a un ou plusieurs spécificateurs d’accès.

# Les classes de l'API java.lang.reflect




BPO11 - Google Slides
-ons Help Last edit was s...
Background
Layout
Theme
8 9 10 11 12 13 14 15


Class
1
*
1
Package

# Que faire avec une instance de la classe java.lang.Class?

```
String s = c.getSimpleName();
        // Nom de la classe
String s = c.getName();
        // Nom complètement qualifié
        // Integer.class  "java.lang.Integer"
        // String[].class  "[Ljava.lang.String"
String s = c.getCanonicalName(); ( Java 5+ )
        // Nom canonique
        // Integer.class  "java.lang.Integer"
        // String[].class  "java.lang.String[]"
Package p = c.getPackage();
         // le package
```

# Consulter les propriétés d'une instance de la classe Class

```
ClassLoader cl = c.getClassLoader();
    // le ClassLoader qui a chargé la classe

boolean b1 = c.isInterface();          // Observer.class
boolean b2 = c.isPrimitive() ;         // int.class
boolean b3 = c.isArray() ;             // Point[].class
    // Connaître la nature du type décrit par la classe

int mod = c.getModifiers() ;
    // Spécificateurs d'accès (abstract, final, …)
boolean isFinal = Modifier.isFinal(mod) ;
    // pour décrypter l'entier fourni par getModifiers()
```

# Consulter les propriétés d'une instance de la classe Class

```
Class<?> sc = c.getSuperClass();
    // la super classe

Class<?>[] ti = c.getInterfaces();
    // les interfaces implémentées
```

# Consulter les fonctions applicables

```
Method[] mp = c.getMethods();
// Méthodes publiques de la classe

Method[] md = c.getDeclaredMethods();
// Méthodes déclarées dans la classe (publiques ou non)


Method mts = c.getMethods(”toString”);
// toString sans paramètre
Class<?> tr = mts.getReturnType();
// le type de retour de la fonction mts
Method mj = c.getMethod(”jouer”, int.class, boolean.class);
// jouer avec 2 paramètres de types int et boolean
```

# Consulter les champs et constructeurs

```
Field[] cp = c.getFields();
Field[] cd = c.getDeclaredFields();
Field c = c.getField(”nom”);


Constructor[] cop = c.getConstructors();
Constructor[] cod = c.getDeclaredConstructors();
Constructor co1 = c.getConstructor();
Constructor co2 = c.getConstructor(int.class, int.class);
```

# Modifier la valeur d'un champ

```
// Hypothèse : le champ abs est public
Point p = new Point(1, 2) ;
p.abs = 256.88 ;

// Utiliser l'introspection pour avoir le même effet
Class<?> c = p.getClass() ;
Field cabs = c.getField("abs");
cabs.setValue(p, 256.88) ;
```

# Modifier la valeur d'un champ


```
// Hypothèse : le champ abs est private
Point p = new Point(1, 2) ;
p.abs = 256.88 ;    // Interdit

// Utiliser l'introspection pour avoir le même effet
Class<?> c = p.getClass() ;
Field cabs = c.getField("abs");
cabs.setAccessible(true) ;
cabs.setValue(p, 256.88) ;
```

→ **Indispensable dans certains frameworks**

# Appeler une fonction

```
Point p = new Point(1, 2) ;
p.deplacer(77., 88.) ;

// Utiliser l'introspection pour avoir le même effet
Class<?> c = p.getClass() ;
Class<>[] tp = {double.class, double.class} ;
Method md = c.getMethod("deplacer", tp) ;
md.invoke(p, 77., 88.) ;
```

→ De même pour les constructeurs

# Créer un objet

```
Point p1 = new Point() ;
Point p2 = new Point(1, 2) ;

// Utiliser l'introspection pour avoir le même effet
Class<?> cp = Point.class ;
Point p1 = (Point)cp.newInstance() ;

Class<>[] tp = {double.class, double.class} ;
Constructor co = cp.getConstructor(tp);
Point p2 = (Point)co.newInstance(77., 88.) ;
```

# … sans utiliser de nom codé en dur

```
Class<?> cp = Class.forName(nomC) ;
Object o1 = cp.newInstance() ;

Constructor co = cp.getConstructor(double.class, int.class);
Object o2 = co.newInstance(77., 4) ;

Method m = cp.getMethod(nomM, double.class, int.class) ;
Object or = m.invoke(o2, 9., 2) ;
```

# Toutes ces fonctions déclenchent des exceptions

**SecurityException**

**NoSuchFieldException**

**NoSuchMethodException**

**InstantiationException**

**IllegalAccessExcepti**

**IllegalArgumentException**

**InvocationTargetException**