

# Introduction à Java

Cours 4: Programmation Orientée Objet en Java

Stéphane Airiau

Université Paris-Dauphine

# Gestion des classes à l'aide de Packages

## (espaces de noms)

## Motivations

---

- garantir l'unicité des noms de classes.

ex : Par exemple, deux développeur peuvent avoir l'idée de développer une classe `Dauphine`. Tant que les classes se trouvent dans des espaces de noms différents, cela ne posera aucun problème et les deux classes pourront être utilisables.

- nombre (parfois important) de classes dans un projet

- ➡ utiliser une structure hiérarchique de répertoires pour ordonner les classes
- ➡ on rassemble dans un package (~ répertoire) les classes reliées.

## Déclaration d'une classe

---

- nom qualifié : nom de la classe à partir de la racine
- ➡ c'est le nom qualifié de la classe qui est unique
- pour les classes de la librairie standard, le nom qualifié commence par `java`, par exemple `java.util.Vector` est le nom qualifié de la classe `Vector`
- Attention, en Java, les espaces de noms ne s'emboîtent pas les uns dans les autres. `java.util` et `java.util.function` sont deux espaces de noms distincts avec chacun leurs classes et interfaces.

Dans le fichier source Java, on commence toujours à indiquer à quel espace de noms la classe appartient en commençant le fichier par la ligne **package** `<nom_du_package>`.

information redondante ➡ on connaît l'emplacement du fichier dans le disque.

**Mais** très utile lorsqu'on lit le code dans un éditeur !

Java vérifie la position du fichier dans le répertoire correspondant

## Utiliser une classe d'un espace de noms

---

- Pour utiliser une classe qui se trouve dans le même espace de noms
  - utiliser son nom simple.
- Pour utiliser une classe à l'extérieur de son espace de noms
  - on peut toujours utiliser son nom qualifié.  
mais cela peut être lourd
  - on fait un **import**  
tout se passe comme si la/les classe(s) importée(s) font partie(s) de l'espace de nom courant.

NB L'import se fait toujours au début de la classe.

## Attention

---

- 1- `import qqch.*`, vous importez seulement les classes, **pas les sous espaces** de noms.
  - ⇒ `import java.*` ne permet pas d'obtenir tous les espaces de noms qui commencent par `java..`
- 2- `import` de plusieurs espaces de noms
  - ⇒ car il est possible d'avoir des conflits de noms

L'utilisation de packages permet alors de désigner sans ambiguïté une classe.



On peut donc avoir deux méthodes avec exactement la même signature tant que ces deux méthodes appartiennent à des espaces de noms différents.

## Librairie standard

---

- `java.lang` contient les classes fondamentales du langage.
- `java.util` contient les classes pour manipuler des collections d'objets, des modèles d'évènements, des outils pour manipuler les dates et le temps, et beaucoup de classes utiles.
- `java.io` contient les classes relatives aux entrées et sorties
- ...

## Compilation et Exécution

---

Pour compiler une classe, il faut indiquer son chemin depuis la racine.

```
| javac important/premier/MaSecondeClasse.java
```

Le compilateur générera le fichier `important/premier/MaSecondeClasse`.  
Si `MaSecondeClasse` possède une méthode `main`, on lance l'exécution en spécifiant le nom complet de la classe. Pour l'exemple, on lancerait donc

```
| java important.premier.MaSecondeClasse
```

# Class Path

### Archivage des classes

➡ utilitaire `jar` qui fait parti de la bibliothèque standard

```
| jar cvf bibliotheque.jar monRepertoireJava/mesClasses/*.class
```

- fonctionne comme le programme `tar` sous unix
- c create an archive
- u mise à jour de l'archive
- x extraction de l'archive
- v verbose
- f use archive file
- e point d'entrée pour une application archivée dans un fichier `jar` exécutable

```
| jar cvfe pro.jar projet.classes.Principale projet/classes/*.class
```

## Dire à Java où chercher

---

Lorsqu'on utilise des fichiers `jar`, il faut dire au compilateur et à la machine virtuelle où se trouvent ces fichiers! ➔ spécifier le `class path`.

Le `class path` peut contenir

- des répertoire qui contiennent des fichiers `.class`
- le chemin de fichiers `jar`
- des répertoires contenant des fichiers `jar`

le compilateur et la machine virtuelle ont une option `-classpath` (abbrégée en `-cp`)

```
java -cp ../libs/lib1.jar projet.classes.Principale
```

Le `class path` aura deux éléments : le repertoire courant `.` et un fichier `jar` nommé `lib1.jar` dans le repertoire `../libs`.

Sous Windows, il faut utiliser des `;` pour séparer les éléments. Sinon, il faut utiliser `:`

Vous pouvez mettre à jour une variable d'environnement `CLASSPATH` sous votre système.

# Annotations

Les annotations sont des balises dans le code Java que des **outils** peuvent utiliser.

ex :

- JUnit : les annotations indiquent des méthodes qui exécutent un test (cf seconde partie du cours)
- JavaPersistence Architecture : relation entre des classes et des tables d'une base de données
- Check Framework : permet par exemple d'ajouter des assertions dans le programme (ex : paramètre est non `null`) ou qu'une chaîne de caractères contient une expression régulière. Un outil d'analyse statique vérifie si les assertions sont valides dans le code.
- outils de la bibliothèque standard pour la compilation

L'objectif aujourd'hui est d'indiquer la syntaxe pour pouvoir lire un code annoté. Nous n'entrerons pas dans les détails mais nous verrons une application avec .

## Annotation

---

- une annotation commence par le caractère @
- @ est suivi par le nom de l'annotation
- en option, on peut avoir des couples (clé,valeur) appelés éléments
- les valeurs peuvent être
  - la valeur d'un type primitif
  - une chaîne de caractères
  - un objet de type `Class`
  - une instance d'un type énuméré
  - une annotation
  - un tableau d'un des types précédents (sauf un autre tableau !)

```
@BugReport (reportedBy={"Barack", "Michelle"})
```

## Annoter des déclarations

---

ces annotations peuvent apparaître lors de la déclaration de

- classes et interfaces
- méthodes
- constructeurs
- variables d'instances
- variables locales
- paramètres de type
- variables de générique
- package

les annotations doivent être déclarées dans une interface d'annotations.

## Annotations standards

Annotation	Applicable à	Buts
Override	Méthode	Vérifie que cette méthode redéfinit une méthode d'une classe parente
Deprecated	toute déclaration	indique que cet élément n'est plus approuvé
SuppressWarnings	toute déclaration (sauf packages)	Supprime les avertissements d'un certain type
Generated	toute déclaration	cet élément a été généré par un outil

Il en existe d'autres que l'on ne verra pas ici (@SafeVarargs, @FunctionalInterface, @Resource, @Resources, @Target, @Retention, @Documented, @Inherited, @Repeatable)

Utiliser @Override est une bonne pratique.

## @SuppressWarnings

---

@SuppressWarnings : Cette annotation supprime des messages d'avertissements pour un certain type.

Si on utilise un cast qui ne peut pas être vérifié au moment de la compilation (ex : `readObject()`), le compilateur Java va écrire un message d'avertissement qui peut être utile.

Mais si on peut garantir qu'il n'y aura pas de problèmes, on peut ignorer cet avertissement.

```
| @SuppressWarnings("unchecked")
```

Parfois la généricité ne permet pas exactement de faire ce que l'on veut (ou le code n'a pas été mis à jour depuis Java5) et on peut ajouter une annotation pour confirmer que ce que l'on fait est correct.

## Tests unitaires et JUnit

L'erreur fait partie de notre nature : il faut donc valider tout travail.  
En plus, on veut éviter que les erreurs passées se reproduisent.

- tests d'intégration : le logiciel fonctionne-t-il dans son environnement d'exécution ?
- tests d'acceptation : le client accepte-t-il le logiciel ?
- tests unitaires : vérifier une "unité" du logiciel ?

On va se concentrer sur les **tests unitaires** :

- aide pour trouver les bugs
- bon moyen de vérifier que tout continue de fonctionner si on a fait des changements

On veut en plus pouvoir automatiser les tests (et éviter d'avoir des appels à `System.out.println(...)` partout!)

- mettre en place les tests peut être coûteux
- s'il n'y a pas d'erreurs, on peut avoir l'impression d'avoir perdu du temps  
(mais réfléchir aux tests peut nous aider à mieux trouver une solution)
- trouver un bug peut prendre beaucoup de temps
- il est impossible/trop coûteux de garantir qu'un code n'a aucun bug  
↳ faire un minimum de tests vérifiant une large partie des cas d'utilisation

Faire une méthode `main` pour tester la classe.

- approche naturelle

**MAIS**

- le test n'est pas forcément documenté
- le test n'est pas forcément compréhensible

⇒ Avoir une classe de tests pour chaque classe développée.

⇒ But : détecter la présence d'une erreur

Attention : passer le test ne garantira pas l'absence d'erreurs.

Pour un cas de test, on a besoin de trois ingrédients

- état initial
- état final
- un outil qui vérifie le résultat théorique au résultat obtenu par le code.

- 
- ne fait pas partie de la bibliothèque standard de Java.
  - à chaque classe, on peut associer une classe de test
  - on peut tester les méthodes que l'on veut
  - squelette :

    créer une instance de la classe à tester

    générer des cas de tests :

- générer les données en entrée
- générer le résultat

    appeler la méthode et vérifier le résultat

Lorsqu'un test échoue, on laisse une explication.

La partie délicate est de faire des tests pour vérifier qu'une expression est bien jetée.

## Définition de tests

---

- une classe de tests unitaire est associée à une classe.
- on ajoute une annotation `@Test` pour indiquer le test
- nom de la méthode quelconque
- visibilité `public`
- retour `void`
- pas de paramètre, peut lever une exception

`if ... fail`

---

La méthode `fail` fait échouer la méthode de test (test non validé).

<code>static void</code>	<code>fail()</code> Fails a test with no message.
<code>static void</code>	<code>fail(String message)</code> Fails a test with the given message.

On a des raccourcis d'écriture avec les méthodes `assert`.

## assert

---

<b>static void</b>	<code>assertTrue(boolean condition)</code> Asserts that a condition is true.
<b>static void</b>	<code>assertNull(String message, Object object)</code> Asserts that an object is null.
<b>static void</b>	<code>assertNotNull(String message, Object object)</code> Asserts that an object is <u>not</u> null.
<b>static void</b>	<code>assertSame(String message, Object expected, Object actual)</code> Asserts that two objects refer to the same object.
<b>static void</b>	<code>assertNotSame(String message, Object unexpected, Object actual)</code> Asserts that two objects do not refer to the same object.
<b>static void</b>	<code>assertEquals(double expected, double actual, double delta)</code> Asserts that two doubles or floats are equal to within a positive delta.
<b>static void</b>	<code>assertEquals(String message, Object expected, Object actual)</code> Asserts that two objects are equal.

On peut exécuter une méthode avant *chaque* test et après chaque test.

- @before par exemple pour créer les instances des états initiaux et finaux, ouvrir des fichiers pour les tests.
- @after pour fermer des ressources ou bien effacer des fichiers tests.

@BeforeClass et @AfterClass fonctionnent de la même façon, mais la méthode est exécutée avant le *premier* test et après le *dernier* test.

## Exemple

---

```
1 package tds.td3;
3 import static org.junit.Assert.*;
5 import java.util.Random;
7 import org.junit.Test;
8
9 public class MyListTest1 {
10
11     @Test
12     public void sizeList() {
13         MyList l = new MyList();
14         l.add("toto");
15         l.add("toto");
16         assertEquals(2, l.size());
17     }
18 }
```

## Exemple (suite)

---

```
19 | @Test
20 | public void sizeList2() {
21 |     MyList l = new MyList();
22 |     l.addLast("z");
23 |     l.add("toto");
24 |     l.add("toto");
25 |     l.addLast("ez");
26 |
27 |     assertEquals(4, l.size());
28 | }
29 |
30 | @Test
31 | public void printList() {
32 |     MyList l = new MyList();
33 |     l.add("toto");
34 |     l.add("toto");
35 |     l.add("titi");
36 |     assertEquals("titi, toto, toto", l.toString());
37 | }
38 |
```

## Exemple (suite)

```
41 | @Test
42 | public void addList() {
43 |     MyList l = new MyList();
44 |     l.add("toto");
45 |     l.add("toto");
46 |     l.add("titi");
47 |     l.addLast("bla");
48 |     l.addLast("bli");
49 |     l.add("doh");
50 |     assertEquals("doh, titi, toto, toto, bla, bli", l.toString());
51 | }
53 |
54 | @Test
55 | public void addList2() {
56 |     MyList l = new MyList();
57 |     l.addLast("toto2");
58 |     l.add("toto");
59 |     l.add("titi");
60 |     l.addLast("bla");
61 |     l.addLast("bli");
62 |     l.add("doh");
63 |     assertEquals("doh, titi, toto, toto2, bla, bli", l.toString());
64 | }
```

## Exemple (suite)

---

```
66  @Test
67  public void getList1() {
68      MyList l = new MyList();
69      l.addLast("toto2");
70      l.add("toto");
71      l.add("titi");
72      l.addLast("bla");
73      l.addLast("bli");
74      l.add("doh");
75      assertEquals("doh", l.get(0));
76  }
77
78  @Test
79  public void getList2() {
80      MyList l = new MyList();
81      l.addLast("toto2");
82      l.add("toto");
83      l.add("titi");
84      l.addLast("bla");
85      l.addLast("bli");
86      l.add("doh");
87      assertEquals("toto", l.get(2));
88  }
```

## Exécution des tests

---

Les tests sont exécutés en utilisant la classe `JUnitCore`.

- en ligne de commande

```
| java org.junit.runner.JUnitCore TestClass [... autres classes de tests ...]
```

- depuis un code Java

```
| org.junit.runner.JUnitCore.runClasses(TestClass.class, ...)
```

## Types énumérés

---

Parfois, on a besoin d'une liste de valeurs possibles

➡ exemple : les tailles des vêtements XS, S, M, L, et XL.

- utiliser ces symboles et leur associer une valeur (par exemple un `final static int`).

Mais on ne pourra pas utiliser directement comme un type `Taille`

➡ **type énuméré**

## enum

---

Pour créer le type énuméré `Size`, on va donc écrire dans le fichier `Size.java` le code ci-dessous :

```
1 public enum Size {  
2     XS,  
3     S,  
4     M,  
5     L,  
6     XL  
7 }
```

## Utilisation

---

- `Size` est un nouveau type
- Les valeurs sont `Size.XS`, `Size.S`, `Size.M`, `Size.L`, et `Size.XL`.
- un type **enum** hérite de la classe `Enum`, qui possède donc des méthodes
  - `values()` retourne la liste de valeurs possibles.
  - `ordinal()` retourne la position de l'instance dans la déclaration du type énuméré

## Exemple

---

```
1 public class Exemple{
2     public static void main(String[] args) {
3         Size mySize = Size.M;
4         for (Size s: Size.values()) {
5             if (s==mySize)
6                 System.out.println("It is my size: "+s);
7             else
8                 System.out.println(s + " is not my size");
9         }
10    }
11 }
```

L'exécution du code précédent donnera :

```
XS is not my size
S is not my size
It is my size: M
L is not my size
XL is not my size
```

## Exemple avec un `switch`

---

```
1 public class Exemple{
2     public static void main(String[] args) {
3         Size mySize = Size.M;
4         double price =0;
5         switch(mySize) {
6             case S: price = 5; break;
7             case M: price = 7; break;
8             case L: price = 9; break;
9             case XL: price = 10; break;
10        }
11        System.out.println("the price is: " + price);
12    }
13 }
```

## Types énumérés

---

Parfois, on a besoin d'une liste de valeurs possibles

➡ exemple : les tailles des vêtements XS, S, M, L, et XL.

- utiliser ces symboles et leur associer une valeur (par exemple un `final static int`).

Mais on ne pourra pas utiliser directement comme un type `Taille`

➡ **type énuméré**

## enum

---

Pour créer le type énuméré `Size`, on va donc écrire dans le fichier `Size.java` le code ci-dessous :

```
1 public enum Size {  
2     XS,  
3     S,  
4     M,  
5     L,  
6     XL  
7 }
```

## Utilisation

---

- `Size` est un nouveau type
- Les valeurs sont `Size.XS`, `Size.S`, `Size.M`, `Size.L`, et `Size.XL`.
- un type **enum** hérite de la classe `Enum`, qui possède donc des méthodes
  - `values()` retourne la liste de valeurs possibles.
  - `ordinal()` retourne la position de l'instance dans la déclaration du type énuméré

## Exemple

---

```
1 public class Exemple{
2     public static void main(String[] args) {
3         Size mySize = Size.M;
4         for (Size s: Size.values()) {
5             if (s==mySize)
6                 System.out.println("It is my size: "+s);
7             else
8                 System.out.println(s + " is not my size");
9         }
10    }
11 }
```

L'exécution du code précédent donnera :

```
XS is not my size
S is not my size
It is my size: M
L is not my size
XL is not my size
```

## Exemple avec un `switch`

---

```
1 public class Exemple{
2     public static void main(String[] args) {
3         Size mySize = Size.M;
4         double price =0;
5         switch(mySize) {
6             case S: price = 5; break;
7             case M: price = 7; break;
8             case L: price = 9; break;
9             case XL: price = 10; break;
10        }
11        System.out.println("the price is: " + price);
12    }
13 }
```