Introduction à la programmation en `Java`

Cours 6

Stéphane Airiau

Université Paris-Dauphine

Class Path

Aider `Java`

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 Javaoù 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 répertoire `../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 sysème.

Annotations

Aider Java

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 Junit .

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 redéfinie une méthode d'une classe parente
`Deprecated`	toute déclaration	indique que cet élément n'est plus approuvé
`SuppressWarnings`	toute déclaration (sauf packages)	Supprimes les avertissements d'un certain type
`Generated`	toute déclaration	cet élément a été généré par un outil

Il en existe d'autres que l'ont 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

Errare Humanum est

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 !)

Coût / Bénéfice

- mettre en place les tests peut être couteux
- 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

Première solution (à bannir)

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

Philosophie

⇨ 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.

Junit

- Junit ne fait pas partie de la bibliothèque standard de Java.
- à chaque classe, on peut associer une classe de test Junit
- 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).

`static void`	`fail()` Fails a test with no message.
`static void`	`fail(String message)` Fails a test with the given message.

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

assert

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

`@Before` et `@After`

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 resources 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.

Test pour vérifier les exceptions

On peut tester si une méthode lance bien une exception (i.e. si la méthode ne lance pas d'exception, il y a un problème).

⇨ on complète l'annotation avec un élément

```
@Test(expected=ArithmeticException)
@Test(expected=expected=IllegalArgumentException.class)
@Test(expected=IllegalStateException.class)
@Test(expected=NullPointerException.class)
```

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    }
```

Exemple (suite)

```
90    @Test(expected=IllegalArgumentException.class)
91    public void getInvalid() {
92      MyList l = new MyList();
93      l.addLast("toto2");
94      l.get(-1);
95    }
96
97    @Test(expected=IllegalArgumentException.class)
98    public void getInvalid2() {
99      MyList l = new MyList();
100     l.addLast("toto2");
101     l.get(10);
102   }
103
```

Exemple (suite)

```
127     @Test(expected=NullPointerException.class)
128     public void sumNull() {
129       MyList l = new MyList();
130       l.add("totoo");
131       l.add(null);
132       assertEquals(5, l.sumLetter());
133     }
134
135     @Test(timeout=1000)
136     public void toStringSpeed() {
137       MyList l = new MyList();
138       for(int i=0;i<100000;i++) {
139         l.add(Integer.toString(i));
140         }
141       l.toString();
142       }
143     }
```

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, ...)
```