

# Introduction aux tests

## DEV 2.3

Florent Madelaine

IUT de Sénart Fontainebleau  
Département Informatique

Année 2021-2022



# Dernier cours sur les tests

- Assertion Java.
- Programmation défensive.

# Assertion Native de Java

- On écrit directement dans le code.  
assert <test>;  
ou si on veut un message détaillé.  
assert <test> : <message>;
- On lance avec l'option -ea (pour *enable assertion*)
- Si on valide le test, on continue après l'assertion
- Sinon l'exécution est interrompue par une AssertionError

## Exemple

```
$ java -ea Liasse
Exception in thread "main" java.lang.AssertionError:
la liasse doit contenir un nombre positif de chaque dénomination
at Liasse.take(Liasse.java:22)
at Liasse.main(Liasse.java:61)
```

## En bref

Les assertions c'est un mode debug.

# Programmation Défensive

Principe qui consiste à restreindre l'utilisation des méthodes, en particulier des méthodes publiques.

**précondition** On restreint les paramètres / valeurs des attributs avant l'appel de la méthode.

On va envoyer des exceptions adaptées

`IllegalArgumentException`,  
`NullPointerException`,  
`IllegalStateException` etc

**invariant** On teste les invariants d'une méthode/ d'une classe.

Normalement `un assert suffit`.

**postcondition** On vérifie après exécution de la méthode que les propriétés adéquates sont satisfaites.

Normalement un `assert` suffit.

# Programmation Défensive

## Pour les méthodes publiques

### Exception vs AssertionError

- Il convient de toujours protéger l'usage d'une méthode publique donc lever une exception dans ce cas.
- Pour tous les autres cas, si le code est correct il n'y a pas d'erreur possible (les invariants et postconditions sont satisfaites). Une assertion suffit.

# Que choisir?

## Point de détail sur les exceptions

**IllegalArgumentException** argument KO (jamais correct)

**IllegalStateException** argument KO (maintenant)

### Exemple

- Jamais le droit de payer avec ma carte sur un terminal chez un marchand.
- Pas le droit aujourd'hui car plafond dépassé.

# Tests unitaires proprement dit

Ce qu'on a vu jusqu'à présent relève plutôt des bonnes pratiques de programmation : programmation par contrat, programmation défensive etc.

On va aussi mettre en oeuvre des **tests unitaire** sur des jeu de données bien choisi. Ceci ne permet pas d'être sûr que le code de la méthode est correct mais ceci permet d'éviter des erreurs fréquentes (condition booléenne écrite à l'envers, erreur de calcul d'indice dans un tableau, inégalité qui devrait être stricte, ...) qui empilées les une sur les autres rendent le debugage particulièrement douloureux.

# Quels tests unitaires?

On peut être plus ou moins exhaustif en fonction du code concerné.

**Méthode** Est-ce que toutes les méthodes ont été testées?

**Choix** Est-ce que les structures de contrôle (if, while, etc) ont été testées pour les 2 cas vrai et faux?

**Chemin** : Est-ce que tous les chemins possibles d'exécution des fonctions d'un programme ont été exécutés?

## En pratique

Il convient surtout d'avoir du bon sens.

Si peu de temps, pas besoin de tester des méthodes simples (get et set qui agissent sans surprise sur un attribut), il vaut mieux se concentrer sur les méthodes qui font quelque chose de plus complexe.

Pour les méthodes plus complexes (ayant besoin d'avoir un diagramme d'activité), il convient de faire a minima un test pour chaque choix et idéalement un test pour chaque chemin.

Chaque test correspond donc à un diagramme de séquence système qui est un cas particulier du diagramme d'activité.



# Tests unitaires avec JUnit

L'inconvénient de faire des tests dans le main, c'est qu'on a du mal à voir quel test est satisfait et lequel ne l'est pas quand on a beaucoup de tests.

Il convient aussi parfois de ne pas toujours lancer un test (par exemple des tests de profilage vérifiant le temps d'exécution moyen d'une méthode qui sont un peu gourmand en temps et en énergie).

JUnit permet de gérer facilement les tests unitaires.

# Quickstart JUnit

## Requis

- Sur votre machine personnelle, il faut installer **JUnit4** : il faut JUnit4 et hamcrest-core (ce sont des jar)  
<https://github.com/junit-team/junit4/wiki/Download-and-Install>
- faire ce qu'il faut à votre CLASSPATH pour que ces jar y apparaissent et que la compilation se fasse sans problème.

## Exemple

À l'IUT il suffit de saisir cette ligne dans votre terminal (ou dans votre .bashrc)

```
export CLASSPATH=".:usr/share/java/junit.jar:usr/share/java/hamcrest-core.jar:$CLASSPATH"
```

# Principe de base en JUnit

- Pour chaque classe java on a un test JUnit (qui est lui aussi un fichier java).
- En général on le nomme plus ou moins pareil que la classe avec Test quelque part dans le nom.
- On a au moins un test pour chaque méthode pas trop simple. (revoir TD semaine dernière sur la notion de couverture).

## Exemple

- Fichier de notre classe : `Calculator.java`
- Fichier testant cette classe : `CalculatorTest.java`

# Hello World en JUnit

## Comment faire tourner les tests?

### En pratique

- 1 compiler les fichiers

```
$ javac *.java
```

- 2 lancer le test

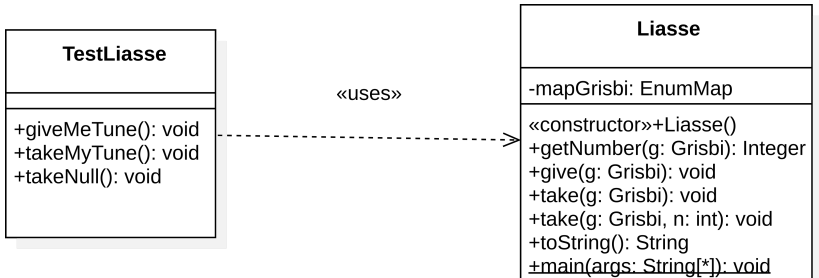
```
$ java org.junit.runner.JUnitCore CalculatorTest0  
JUnit version 4.12
```

```
.
```

```
Time: 0.004
```

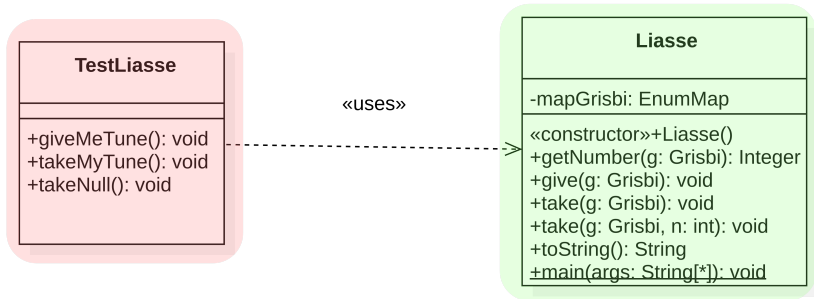
```
OK (1 test)
```

# Junit sur une image



On exécute la **classe de test**, qui elle va appeler la classe qu'on souhaite tester.

# Junit sur une image



On exécute la **classe de test**, qui elle va appeler la classe qu'on souhaite tester.

# Hello World en JUnit

## Fichier Calculator.java

```
/**
 * Calculator est une classe offrant une seule méthode qui évalue une
 * somme, donnée sous la forme d'une chaîne de caractère listant des
 * opérandes séparées par des +
 */
public class Calculator {
    /**
     * somme les opérandes passées sous forme d'une chaîne de
     * caractères et retourne le résultat sous forme d'entier.
     * @param expression : chaîne de caractères
     * ("nombres" séparés par des + sans espaces).
     * Par exemple "42+3" ou encore "-42+42"
     * (le moins unaire est autorisé).
     * plus spécifiquement nombre est à comprendre au sens de
     * parseInt(java.lang.String)
     * @throws NumberFormatException :
     * si l'expression n'est pas dans ce format
     * (par exemple "x+2" ou " 1 +2" -- il y a des espaces --
     * ou encore "99999999990").
     */
    public int evaluate(String expression) {
        int sum = 0;
        for (String summand: expression.split("\\+"))
            sum += Integer.valueOf(summand);
        return sum;
    }
    /**
     * Pour appeler cette super méthode depuis la ligne de commande
     * (on ne regarde que le premier argument, les autres sont ignorés).
     */
    public static void main(String[] args) {
        Calculator calculator = new Calculator();
        System.out.println(calculator.evaluate(args[0]));
    }
}
```

# Hello World en JUnit

## fichier CalculatorTest

```
import static org.junit.Assert.assertEquals;
import org.junit.Test;

public class CalculatorTest {

    // un test pour Junit4 c'est une méthode avec l'annotation suivante devant la méthode.
    @Test
    public void evaluatesGoodExpression() {
        Calculator calculator = new Calculator();
        int sum = calculator.evaluate("1+2+3");
        // on peut stipuler que des choses sont normalement égales
        // charger de manière statique les Assert si on veut éviter d'avoir à écrire de quelle classe on parle)
        assertEquals(6, sum);
    }
}
```



# En pratique pour le TP

- Chaque test qu'on avait fait manuellement dans le main devient une méthode dans une classe de test.
- Cette méthode porte un nom bien choisi indiquant ce qu'elle teste (la documentation de ce test devient presque toujours inutile).

## À éviter

test0, test1, test2...

Si test2 échoue, on ne sait pas de quoi il s'agit!

# Exemples plus complexes d'utilisation de Junit

Vous pouvez regarder le code que j'ai tiré de la documentation de junit4 qui est disponible sur eprel.

**Lisez sur eprel**

`Junit4Exemples.tar.gz`

## Qu'est-ce-qui fait un bon test

Maintenant que nous avons entrevu les aspects techniques du test avec Junit. Il faut que nous réfléchissions à **quels tests nous devons écrire**.

Tester correctement n'est pas une tâche simple. Comme la programmation, c'est aussi une activité qui demande de l'entraînement et un peu de réflexion préalable.

Nous avons déjà évoqué la semaine dernière qu'il fallait essayer de **couvrir toutes les branches d'une méthode importante**.

On peut toutefois couvrir toutes les branches avec des tests sans intérêt. Il convient donc d'en fabriquer qui sont pertinents.

## Right BICEP

Jeff Langr, Andy Hunt et Dave Thomas, les auteurs de *Pragmatic Unit Testing in Java 8 with JUnit*, proposent un cadre simple basé sur l'acronyme **Right BICEP**.

- Right** Are the results right?
- B** Are all the boundary conditions CORRECT?
- I** Can you check inverse relationships?
- C** Can you cross-check results using other means?
- E** Can you force error conditions to happen?
- P** Are performance characteristics within bounds?

### Slogan

“Utilise ton biceps droit!”

## Right BICEP

Jeff Langr, Andy Hunt et Dave Thomas, les auteurs de *Pragmatic Unit Testing in Java 8 with JUnit*, proposent un cadre simple basé sur l'acronyme **Right BICEP**.

- R**ight Are the results right?
- B** Are all the boundary conditions CORRECT?
- I** Can you check inverse relationships?
- C** Can you cross-check results using other means?
- E** Can you force error conditions to happen?
- P** Are performance characteristics within bounds?

### Slogan

"Use your Right BICEP"

# Right : que doit faire la méthode?

## are the results right?

Ce sont les tests les plus simples à écrire normalement. A priori ils sont implicites dans les spécifications.

- Si ce n'est pas clair, il faut demander au client ou au responsable du projet qui a vu le client.
- En attendant, il faut réfléchir vous-même à ce que vous pensez être la bonne spécification.
- De toute façon la spécification (et donc les tests) risquent de changer en cours de projet.

### dh'o

Pas possible de coder la méthode si on n'a pas une petite idée de ce qu'elle doit faire.

## B : les cas limites.

### Are all the boundary conditions CORRECT?

On teste une forme de robustesse de la méthode.

Ce sont souvent les cas qui ne doivent pas être traités normalement : entrée incohérente ou non conforme ou encore une entrée du type correct mais en dehors des valeurs attendues.

Pour des structures de données sous-jacentes qui sont ordonnées, on regarde ce qui se passe pour des cas particuliers concernant le premier et le dernier élément (erreur borne parcours).

## B : les cas limites

### Exemples

- fichier avec un nom invraisemblable  
`xyz#"\/toto^~#- ([°`
- adresse mail pas bien formée  
`toto.titi.tata@com`
- valeur manquante  
`null`
- type correct mais valeur pas adaptée (e.g. taille d'une personne de 3m)



# B : les cas limites.

## Are all the boundary conditions CORRECT?

En fait ce cas est une sorte de cas un peu fourre tout pour lequel Jeff Langr, Andy Hunt et Dave Thomas, proposent un acronymes de plus **CORRECT**.

### Conformance

Order

Range

Reference

Existence

Cardinality

Time

### Idée générale

Pour les données en entrée de la méthode ou maintenues en internes pour la classe, **qu'est-ce-qui pourrait mal se passer?**

## Can you check inverse relationships?

Certaines méthodes ont un inverse naturel facile à vérifier.

### Exemples

- racine carré et produit.
- inverse modulaire et produit modulaire.

### À éviter

ne pas coder les 2 méthodes soit-même.

# C

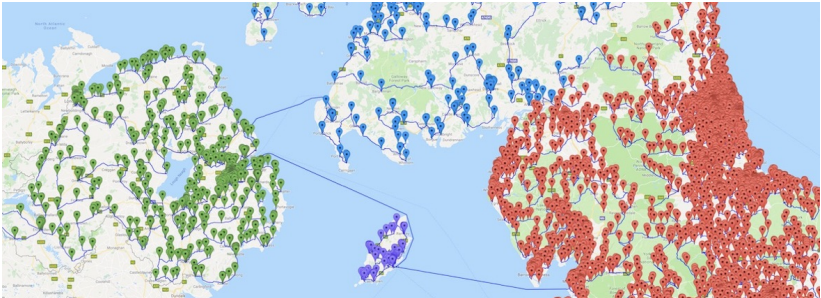
## Can you cross-check results using other means?

Idée similaire mais plus générale que la précédente.

### Exemple

- Utilisez une méthode intelligente pour résoudre le voyageur de commerce.
- Comparer avec la méthode bête qui teste toutes les permutations de villes possibles.

# Digression : le voyageur de commerce



*pub crawl* des 24,727 pubs du Royaume-Uni.

<http://www.math.uwaterloo.ca/tsp/uk/index.html>

# E

## Can you force error conditions to happen?

Force ou simule des erreurs pour voir si l'appli tient le coup (disque plein, coupure réseau, bdd inaccessible, problème d'horloge etc).

Dépasse le cadre de ce cours. En pratique, on ne vas pas le faire réellement. On peut même le faire de manière interne à java avec les **Mock Objects**.

# P

## Are performance characteristics within bounds?

On étudie par exemple l'efficacité de la méthode avec l'augmentation de la taille de l'entrée.

### Exemple

Est-ce que ma super méthode théoriquement linéaire sur les graphes l'est vraiment en pratique si je passe de 1000 sommets à 100 000 sommets?

### Remarque

Pour aller plus loin il y a des outils dédiés come JUnitPerf.

# Calcul du plus grand entier dans une liste

```
public class Largest {  
    /**  
     * Return the largest element in a list.  
     *  
     * @param list A list of integers  
     * @return The largest number in the given list  
     */  
    public static int largest(int[] list) {  
        int index, max=Integer.MAX_VALUE;  
        for (index = 0; index < list.length-1; index++)  
            if (list[index] > max) {  
                max = list[index];  
            }  
        }  
        return max;  
    }  
}
```

## Remarque finale

Pour éviter les bugs liés au parcours de liste etc, il vaut mieux éviter les index manuels et utiliser la facilité java des boucles permettant d'utiliser l'interface `iterable`.

### Exemple

itérer sur un type énuméré.

```
for (Grisbi g : Grisbi.values())
    System.out.printf("%s est le grisbi de dénomination %d%n",
                      g, g.getDenomination());
```



# Bibliographie

- documentation de Junit4.
- *Pragmatic Unit Testing in Java 8 with JUnit* par Jeff Langr, Andy Hunt et Dave Thomas.