Ing39: Tests statiques avec JUnit4

M. V. Aponte
G. Levieux
S. Rosmorduc

Documentation en Javadoc

- but : fournir une documentation toujours à jour des méthodes d'une classe

- pratique courante dans la plupart des langages de programmation: documenter une procédure ou une fonction à l'aide de commentaires

- en pratique: formalisme permettant d'extraire *automatiquement* une documentation à partir des sources java

Programmation par contrat

- Idée importante: un sous-programme passe un « contrat » avec son utilisateur:
  - « si tu me passe tel ou tel argument, alors je ferai/je renverrai tel ou tel résultat »
- On y spécifie :
  - ce qui doit être vrai au moment de l'appel
  - ce qui est vérifié après l'appel
- ces informations doivent figurer dans la documentation du sous-programme.

Programmation par contrat

- pré-conditions: conditions qui doivent être remplies pour appeler le sous programme
  - généralement contraintes sur les arguments
  - si elles ne sont pas remplies, normalement: levée d'exception
- post-condition: conditions qui doivent être remplies après l'appel du sous programme. Elles précisent **ce que fait** le sous-programme
- Un contrat est une documentation précise d'un sous-programme!

Exemple pour une méthode

public static double moyenne(double [] tab)

- pré-condition: tab n'est ni null, ni vide (tab.length > 0)
- post-condition: la valeur retournée est la moyenne des valeurs comprises dans tab

- Attention: si la pré-condition n'est pas respectée, le contrat ne dit pas ce que fait la méthode. Son comportement reste *non spécifié* dans ce cas.

- On utilisera un commentaire javadoc pour spécifier les contrats.

Le contrat en javadoc

```
/**
 * Un ensemble de méthodes utilitaires pour travailler sur des tableaux.
 */
public class TableauxUtils {

    /**
     * Calcule la moyenne des éléments d'un tableau.
     * <p> Pré-condition : le tableau ne doit être ni null,
     * ni vide (tab.length &gt; 0) </p>
     * @param tab un tableau non vide
     * @return la moyenne des valeurs du tableau
     * @throws IllegalArgumentException si tab est null ou de taille 0
     */
    public static double moyenne(double tab[]) {
```

Tests et JUnit 4

Tests

- But : détecter *de manière automatique* des erreurs dans les programmes

- Normalement, un test réussi ne prouve rien… mais il rassure quand même

- Un test qui échoue montre un bug

- En pratique, c'est un outil extrêmement important pour développer des logiciels robustes.

- Les tests sont automatisés avec JUNIT

Tests unitaires / test fonctionnels

- **test unitaire**: on teste qu'un sous programme fonctionne comme dit dans son contrat (fonctionnement correcte).
    - c.a.d, que si on l'appelle en respectant ses pré-conditions, alors ses post-conditions seront vérifiées.

- **test fonctionnel**: on teste un scénario, composé de plusieurs appels de sous programmes

Test de non régression

- **non régression**: on conserve tous les tests ayant déjà servi et on les rejoue quand le logiciel évolue.

- on peut ainsi détecter des régressions: du code qui fonctionnait, et qui ne fonctionne plus

- les tests permettent de faire évoluer le logiciel en détectant et corrigeant très tôt les régressions

Test-driven development (très simplifié)

TDD ou ***développement piloté par les tests***

1. On commence par définir les interfaces des sous-programme (leur en-tête) et leurs contrats

2. Ensuite on écrit les tests que ces sous programmes doivent réussir

3. Ensuite seulement, on écrit le code des sous programme

4. tester, recommencer jusqu'à ce que tous les tests passent

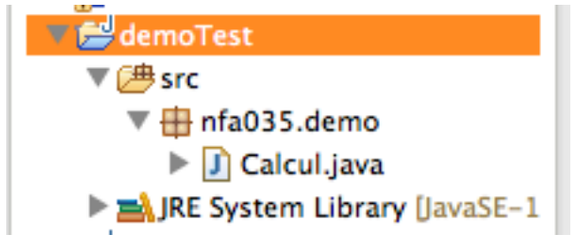
Notre classe à tester…

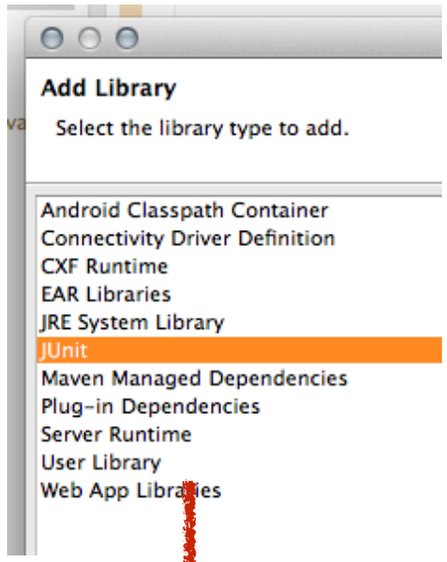
```java
package nfa035.demo;

public class Calcul {

  /**
   * Calcule a puissance n.
   * @param a un entier
   * @param n un entier positif ou nul.
   * @return a puissance n
   * @throws IllegalArgumentException si n n'est pas positif ou nul.
   */
  public static int puissance(int a, int n) {
   if (n < 0) throw new IllegalArgumentException();
   int resultat= 0;
   for (int i= 0; i < n ; i++)
      resultat= resultat* resultat;
   return resultat;
  }
}
```

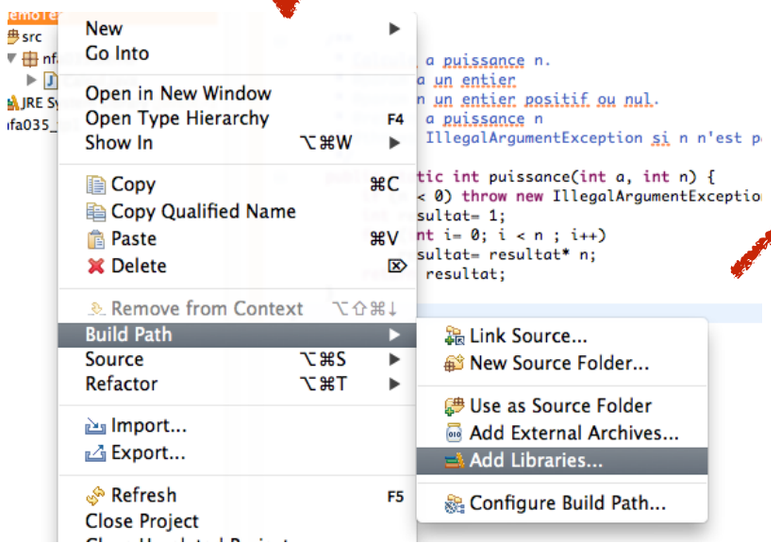
(attention, buggée!!!)

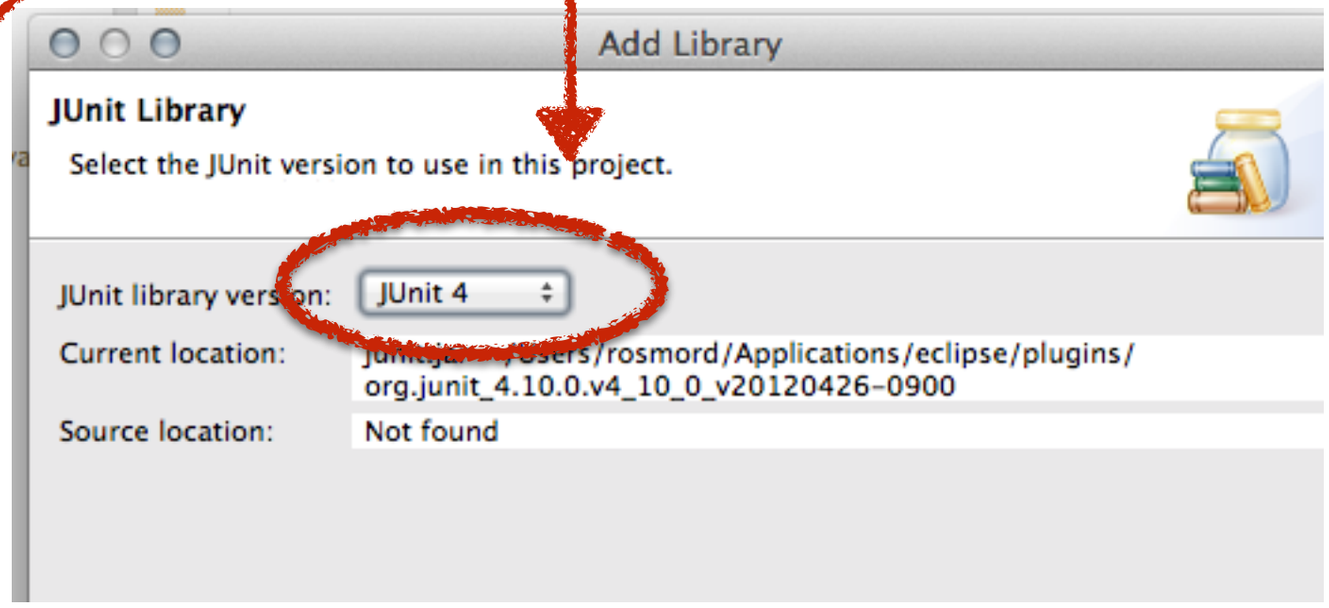
La première fois

- Ajouter JUnit4 à son projet:
- `BuildPath -> Add Library -> Junit -> Junit4`



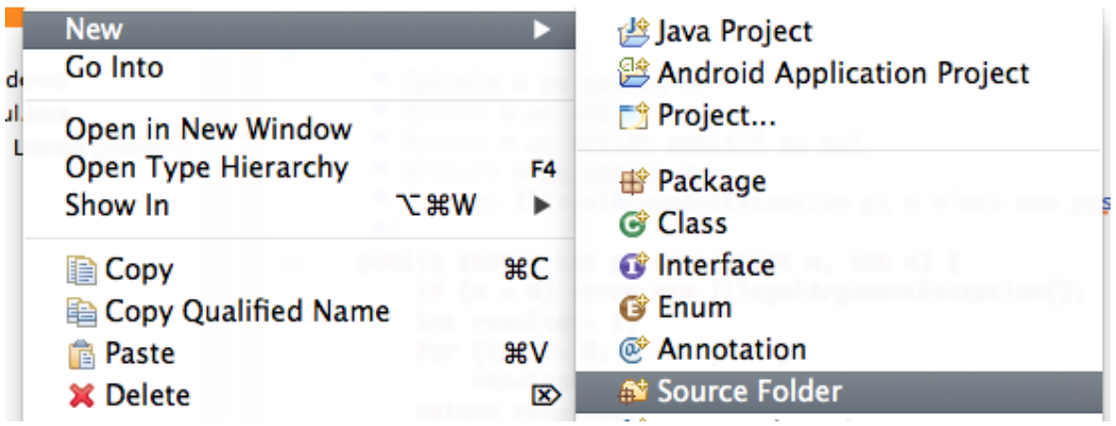


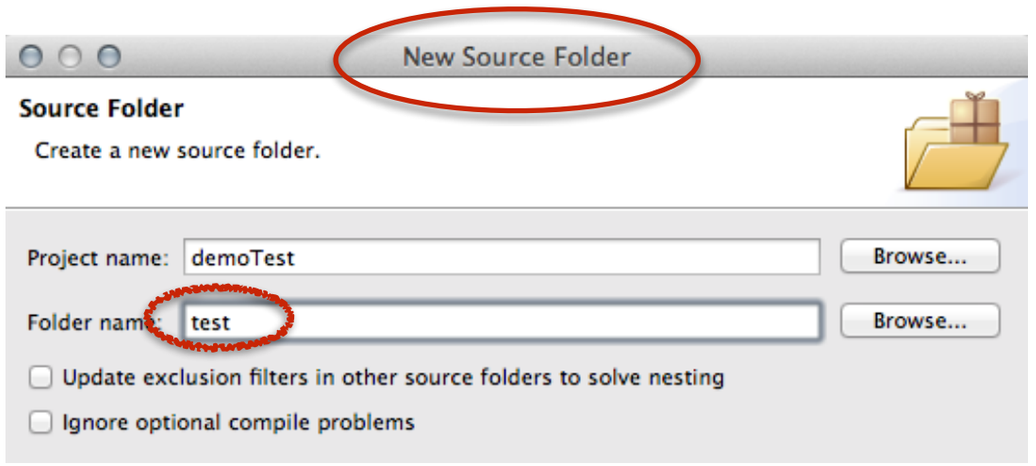


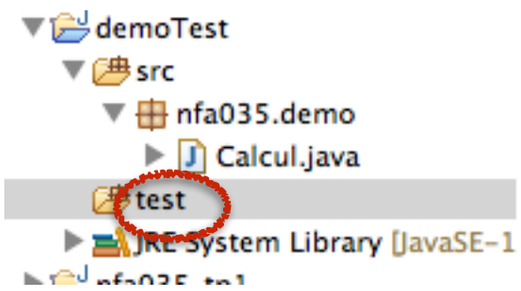


(Optionnel mais fort utile)

- Créer un dossier de type « Source Folder » pour les sources des tests:







Par la suite

- Créer une nouvelle classe *(New -> Unit Test Case)* dite classe de tests. Son nom est celui de la classe à tester suivit du mot Test.

- Elle est générée avec annotations Junit. Vous devez y ajouter des méthodes pour tester les méthodes de **`Calcul.java`**.

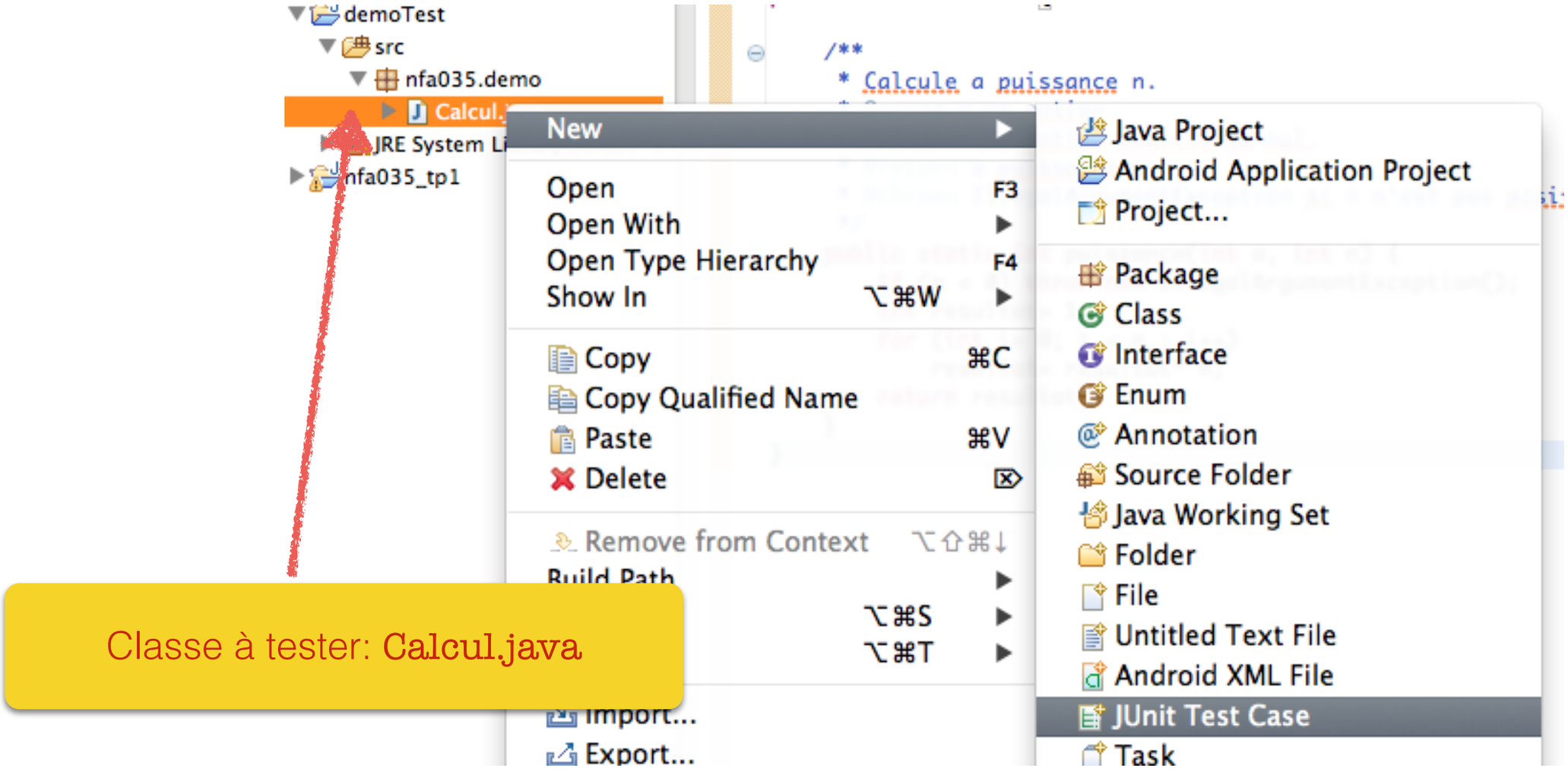
- Placer **`CalculTest.java`**, dans le répertoire des tests et dans le même paquetage que **`Calcul.java`**.

- Exécuter **`CalculTest.java avec Junit4.`** *(Run as Junit Test Case)* Se traduit par l'appel une à une des méthodes de tests.

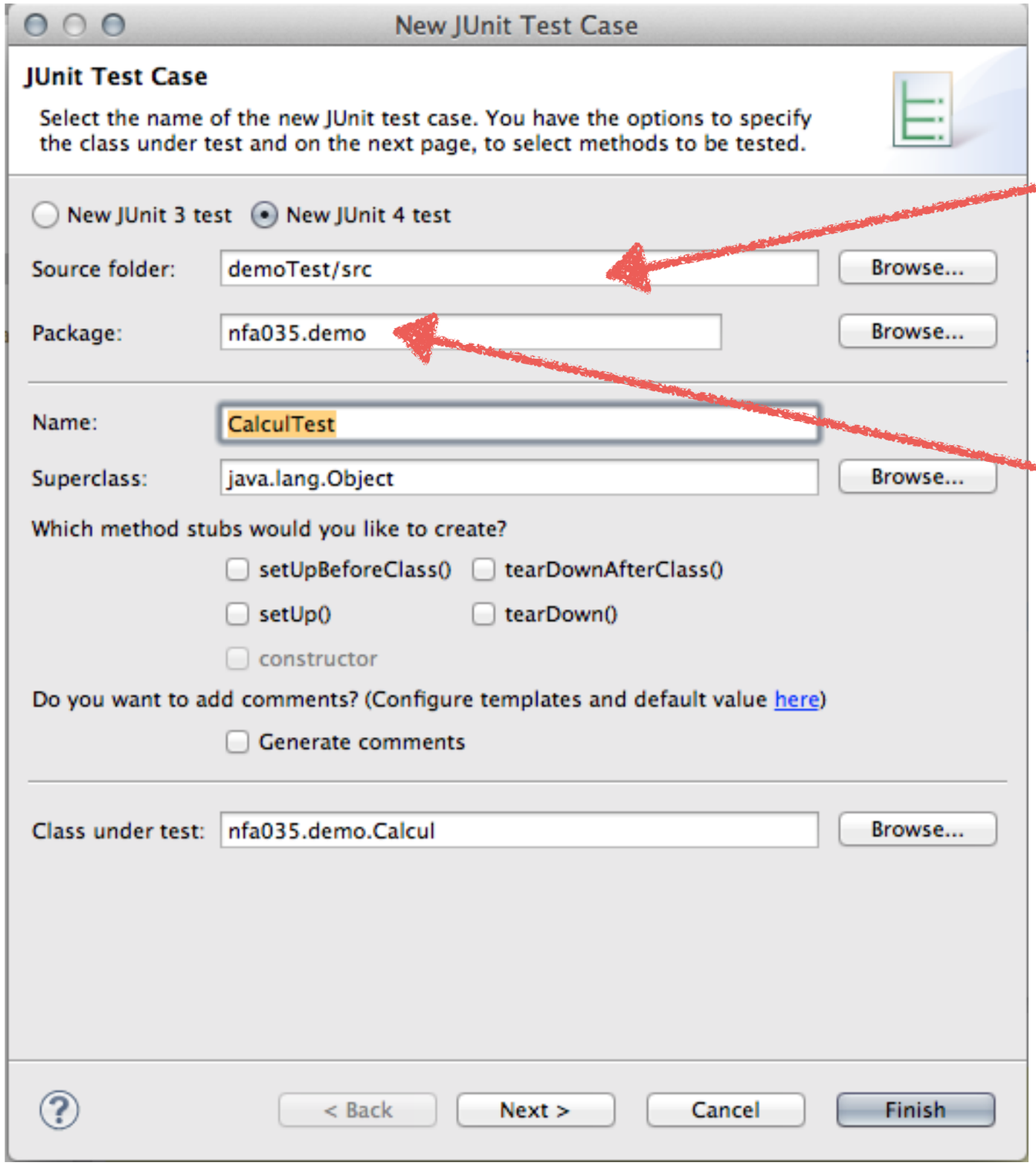
- Junit4 produit un rapport indiquant les méthodes de test qui ont réussi et celles qui ont échoué.

Création de la classe de test

- On sélectionne la classe à tester, et on crée un « cas de test »
- cela produit une nouvelle classe **CalculTest.java** contenant les sources pour jouer les tests



Création de la classe de test

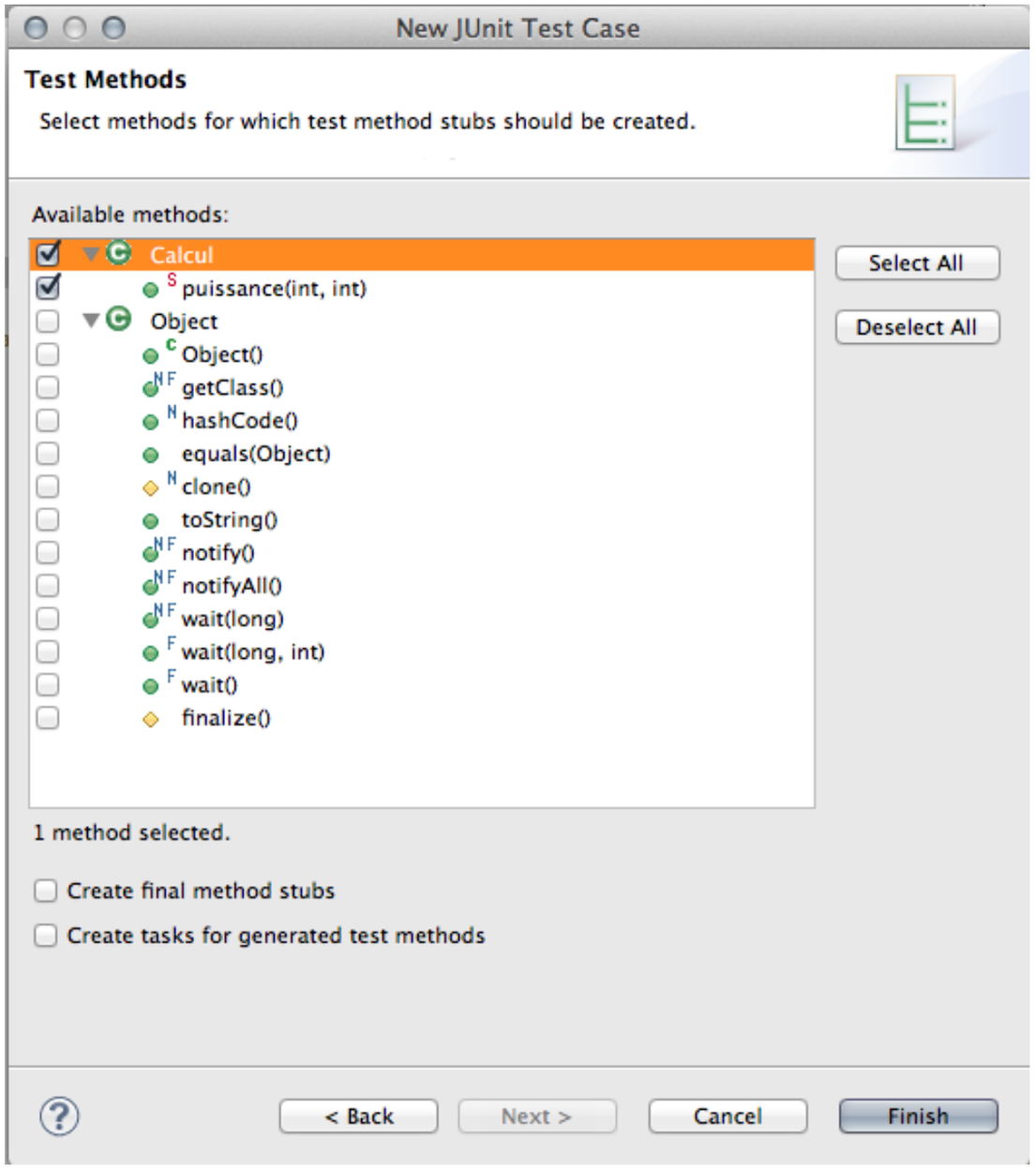


- Nom par défaut de la classe des tests: `CalculTest.java`
- Emplacement par défaut: le même que la classe à tester (`demoTest/src`)
- Modifier éventuellement cet emplacement pour le dossier `demoTest/tests`

ATTENTION: garder le même package que celui de la classe à tester.

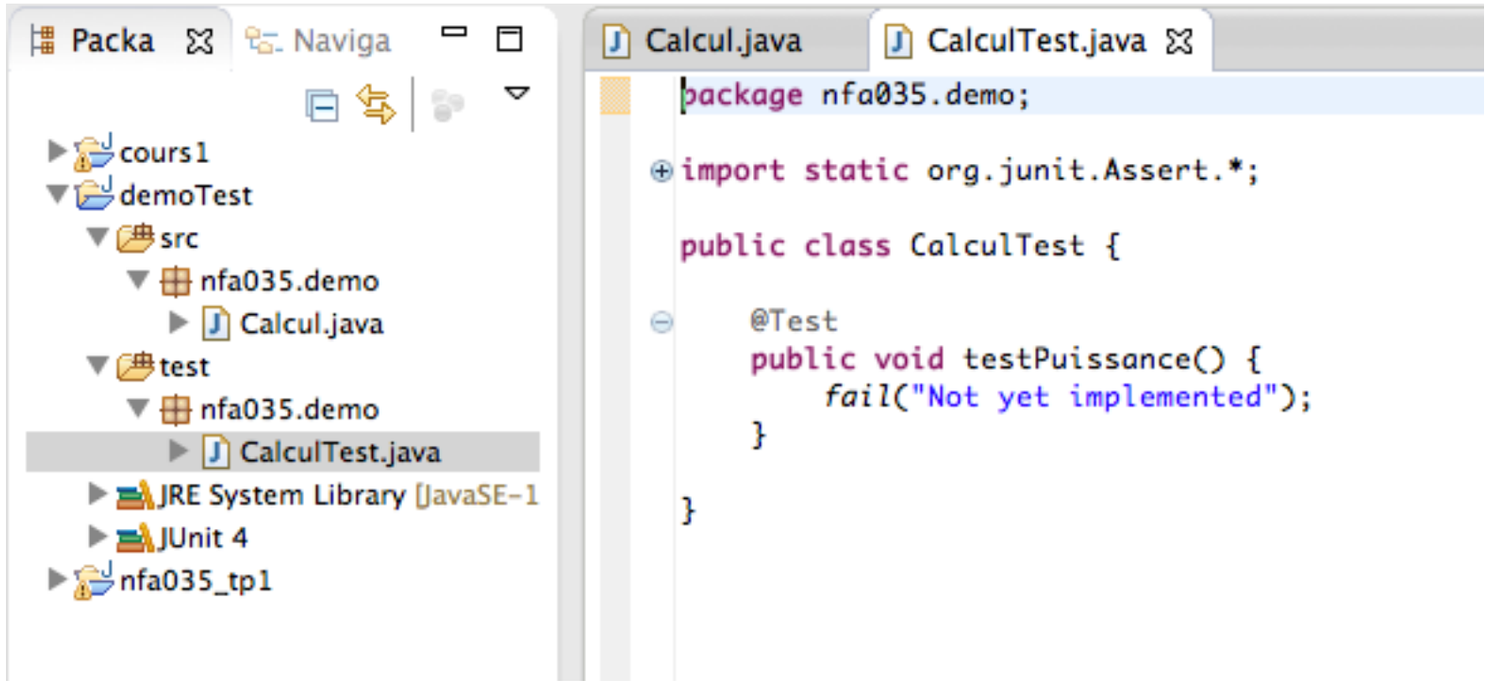
Création des tests

- Choisir les méthodes à tester (optionnel, on peut facilement ajouter des tests après coup)



Création des test

La classe de test est normalement mise dans le même package que la classe à tester



La classe des tests

- La classe de test comporte normalement ces deux **import**:

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

- Elle est composée de méthodes de tests:
  - méthode Java précédée de l'annotation **@Test**
  - son but: tester **UN** des cas d'appel pour **UNE** des méthodes à tester.

Cas de test

- Correspond à un cas concret d'appel d'une méthode à tester. Ex: pour tester la méthode

```
static int puissant(int a, int n){ ...}
```

- on devra l'appeler sur des arguments concrets puis comparer le résultat obtenu avec ce que dit son contrat pour ce cas là. Par exemple:

  appeler **`puissance(3,1)`** **qui** doit renvoyer **`3`**
  appeler **`puissance(0,0)`** qui doit renvoyer **`1`**
  appeler **`puissance(0,-1)`** **qui** doit lever IllegalArgumentException,
  … etc

- Concrètement, une méthode de test doit :

  1. Appeller la méthode à tester avec des arguments concrets

  2. Vérifier avec les méthodes de la classe Assert que le résultat est conforme avec ce que dit le contrat.

- On écrit une méthode de test par cas d'appel à effectuer (donc, plein de cas de test)!

Un cas de test: $2^3 = 8$

Annotation @Test

```
package nfa035.demo;

import org.junit.Assert;
import org.junit.Test;

public class CalculTest {
  @Test
  public void testPuissanceN() {
     Assert.assertEquals(8, Calcul.puissance(2, 3));
  }
```

puissance(2,3) doit renvoyer 8

assertEquals:
premier argument: valeur attendue
second argument: valeur calculée
test réussi si les deux sont égales

Appel au code à tester : méthode puissance de la classe Calcul

Quels tests écrire ?

- Tester le « cas général » : définir ce qu'on attend, le calculer dans quelques cas, et vérifier qu'on a bien ce qui était attendu

- Tester les « cas limites » : cas où on s'attend à des problèmes, par exemples valeurs extrêmes… ici, 0

- Tester les conditions d'erreur: vérifier que les mauvais paramètres sont détectés, et correctement traités (engendrent des exceptions)

- après découverte de bug:

  - créer un test pour mettre le bug en évidence

  - quand le bug est corrigé, le test « passe »

Vérification des exceptions

- Dans les cas où un sous-programme doit échouer avec une exception, on écrit :

- **@Test(expected= *ClasseDeLException*.class)**

- Exemple

```
@Test(expected= IllegalArgumentException.class)
public void testPuissanceNegative() {
   int res= Calcul.puissance(2, -3);
}
```

Protection contre les boucles infinies

- On peut donner un temps maximal à un test pour tourner :

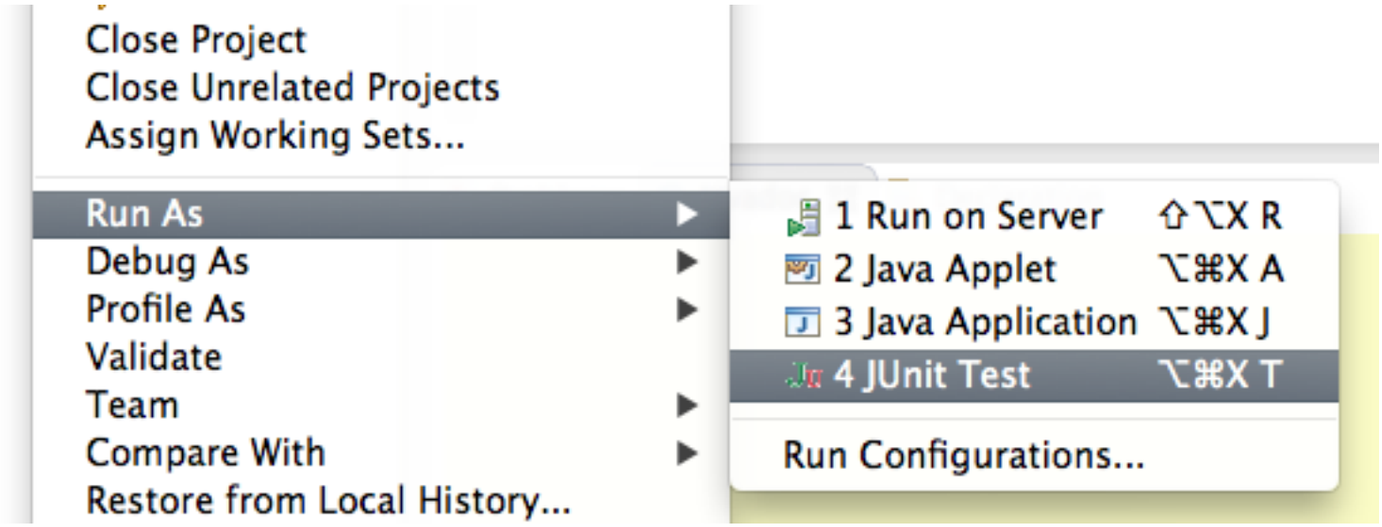
@Test(timeout=1000)

- le temps est donné en milliseconde
- ainsi, en cas de boucle infinie, le test échoue

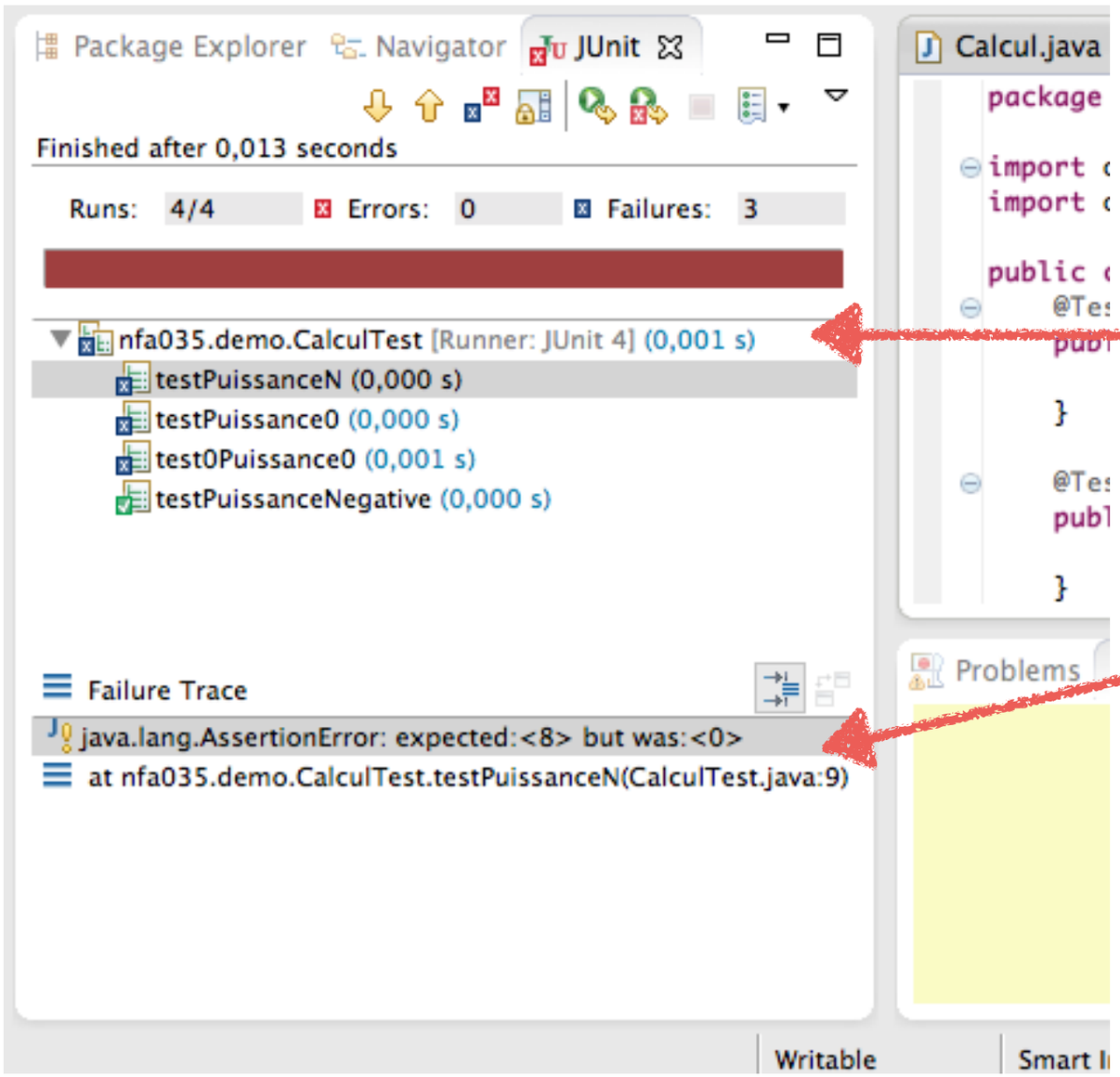
Les tests pour notre classe

```
public class CalculTest {
  @Test(timeout=1000)
  public void testPuissanceN() {
   Assert.assertEquals(8, Calcul.puissance(2, 3));
  }
  @Test(timeout=1000)
  public void testPuissance0() {
   Assert.assertEquals(1, Calcul.puissance(1000, 0));
  }
  @Test(timeout=1000)
  public void test0Puissance0() {
   Assert.assertEquals(1, Calcul.puissance(0, 0));
  }
  @Test(expected= IllegalArgumentException.class, timeout=1000)
  public void testPuissanceNegative() {
    int res= Calcul.puissance(2, -3));
  }
}
```

Faire tourner les tests dans eclipse



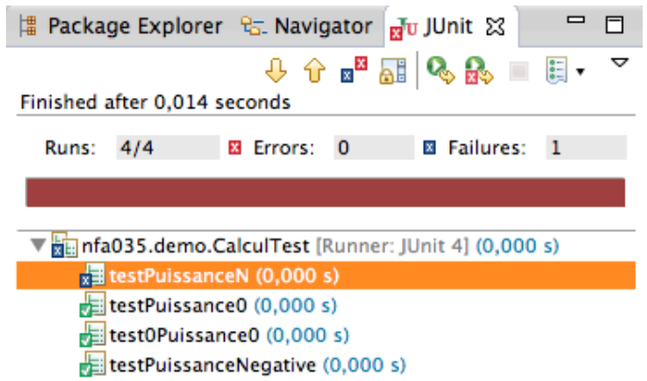
Faire tourner les tests dans eclipse

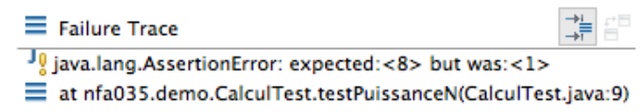


- 4 tests, 3 échecs…
- sur le premier : on attendait « 8 », mais le résultat fut 3

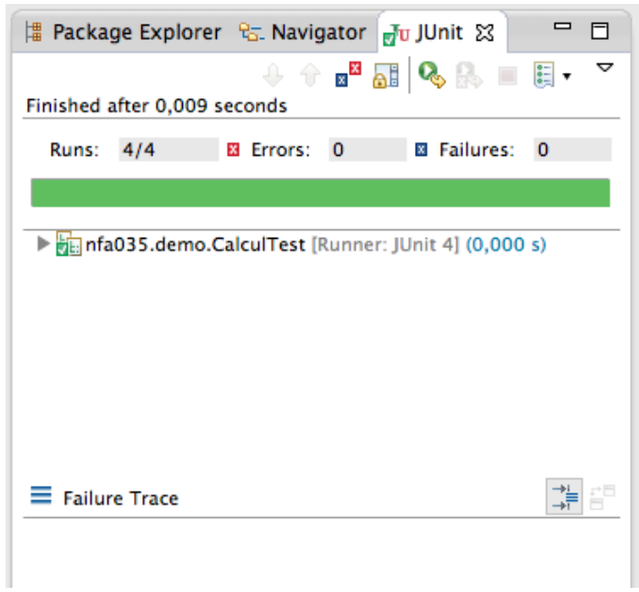
On corrige…

- Ça doit être resultat= 0 au début… il faudrait l'initialiser à 1…





- plus qu'une erreur… ah, oui, resultat= resultat* resultat c'est faux… on doit mettre resultat= a*resultat



- barre verte= tous les tests sont bons

Quelques méthodes de Assert…

- Assert.assertEquals(valAttendue,valCalculée) :
  - test qui échoue si valCalculée != valAttendue
- Assert.assertArrayEquals(valAttendue,valCalculée)
  - même chose, mais les valeurs sont des tableaux
- Assert.assertSame(valAttendue,valCalculée) :
  - comme assertEquals, mais compare les adresses des objets (n'utilise pas equals())
- Assert.assertTrue(condition)/Assert.assertFalse(condition)
- Assert.fail() : l'exécution de cette méthode fait échouer le test

Messages dans Assert

- Toutes les méthodes ont une variante qui prend en plus un premier argument qui est un message à afficher en cas d'erreur. Il décrit typiquement le test

```
@Test(timeout=1000)
public void testPuissanceN() {
   Assert.assertEquals("calcul de 2^3",8, Calcul.puissance(2, 3));
}
```

double, float et tests

- Les calculs sur les nombres réels sont **approchés** (voir https://fr.wikipedia.org/wiki/IEEE_754)
- Exemple :

```
int n= 10;
double x= 1.0/n;
double s= 0.0;
for (int i= 0; i < n; i++) {
 s+= x;
}
System.out.println(s);
System.out.println(s  == 1.0);
```

oui, même 1/10 est approché !
(l'ordinateur travaille en base 2)

```
0.9999999999999999
false
```

(essayez ce code avec n=4096, une puissance de 2, et vous aurez un résultat exact)

double, float et tests

- Les calculs sur les nombres réels sont **approchés** (voir https://fr.wikipedia.org/wiki/IEEE_754)
- Pour tester un résultat réel, on doit se donner une **marge d'erreur** :
  - au lieu de tester que Math.cos(0) == 1.0, on teste par exemple que

$$|\texttt{Math.cos(0)} - 1.0| < 0.00001$$

  - avec JUnit, ça donne

```
assertEquals("test cos", 1.0, Math.cos(0), 0.00001);
```

message

valeur attendue

valeur calculée

marge d'erreur

import static

- fonctionnalité de java 1.5 (et plus) : au lieu de taper à chaque fois **Assert.*nomMethode*** on peut utiliser un import static:

import static org.junit.Assert.*;

- on peut ensuite utiliser les méthodes directement:

assertEquals("calcul de 2^3",8, Calcul.puissance(2, 3));

- *(personnellement, je trouve que ça rend la lecture des classes plus complexes. Pour JUnit, c'est pratique, mais en général, on ne sait plus trop d'où viennent les méthodes… mais ça n'engage que moi. S.R.)*