NFA035 : Javadoc et JUnit

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

- idée : donner un formalisme qui permette d'extraire automatiquement une documentation structurée 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 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
  - précisent ce que fait le sous-programme

Exemple

- 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

Javadoc en pratique

- Commentaire « javadoc » de la forme

```
/**
 *
 */
```

- Avant la classe : documentation de la classe toute entière

- Avant la méthode : documentation de la méthode

Un exemple concret

```
/**
 * 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[]) {
```

Structure du commentaire javadoc

- première ligne, terminée par un point: résumé de ce que fait la méthode
- suite : texte documentant la méthode. On peut utiliser des balises HTML
- ensuite: annotations pour décrire:
  - les paramètres de la méthode : @param
  - la valeur de retour @return
  - les exceptions @throws

```
/**
 * 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[]) {
```

Résumé (1 ligne terminée par « . »)

documentation

Description d'un paramètre

valeur de retour

Exceptions

Lire la javadoc

Lire la javadoc

Overview Package **Class** Use Tree Deprecated Index Help

Java™ Platform Standard Ed. 7

Prev Class Next Class Frames No Frames

Summary: Nested | Field | Constr | Method Detail: Field | Constr | Method

java.lang

Class String

java.lang.Object
  java.lang.String

All Implemented Interfaces:

Serializable, CharSequence, Comparable<String>

Position de cette classe dans la hiérarchie

```
public final class String
extends Object
implements Serializable, Comparable<String>, CharSequence
```

Introduction (souvent assez détaillée)

The `String` class represents character strings. All string literals in Java programs, such as `"abc"`, ar
instances of this class.

Strings are constant; their values cannot be changed after they are created. String buffers support mutable strings.
Because String objects are immutable they can be shared. For example:

```
String str = "abc";
```

Methods

Modifier and Type	Method and Description
char	**charAt**(int index) Returns the char value at the specified index.
int	**codePointAt**(int index) Returns the character (Unicode code point) at the specified index.
int	**codePointBefore**(int index) Returns the character (Unicode code point) before the specified index.
int	**codePointCount**(int beginIndex, int endIndex) Returns the number of Unicode code points in the specified text range of this String.
int	**compareTo**(String anotherString) Compares two strings lexicographically.
int	**compareToIgnoreCase**(String str) Compares two strings lexicographically, ignoring case differences.
String	**concat**(String str) Concatenates the specified string to the end of this string.
boolean	**contains**(CharSequence s) Returns true if and only if this string contains the specified sequence of char values.
boolean	**contentEquals**(CharSequence cs) Compares this string to the specified CharSequence.
boolean	**contentEquals**(StringBuffer sb)

index des méthodes et des champs

détail sur une méthode

indexOf

```
public int indexOf(String str)
```

Returns the index within this string of the first occurrence of the specified substring.

The returned index is the smallest value *k* for which:

```
this.startsWith(str, k)
```

If no such value of *k* exists, then `-1` is returned.

Parameters:

`str` - the substring to search for.

Returns:

the index of the first occurrence of the specified substring, or `-1` if there is no such occurrence.

Que fait-elle ?

Spécification

Cas particuliers

Paramètres

Valeur de retour, et cas particulier

Autre exemple

substring

```
public String substring(int beginIndex,
                        int endIndex)
```

Returns a new string that is a substring of this string. The substring begins at the specified `beginIndex` and extends to the character at index `endIndex - 1`. Thus the length of the substring is `endIndex-beginIndex`.

Examples:

```
"hamburger".substring(4, 8) returns "urge"
"smiles".substring(1, 5) returns "mile"
```

Parameters:

`beginIndex` - the beginning index, inclusive.

`endIndex` - the ending index, exclusive.

Returns:

the specified substring.

Throws:

`IndexOutOfBoundsException` - if the `beginIndex` is negative, or `endIndex` is larger than the length of this `String` object, or `beginIndex` is larger than `endIndex`.

Cas d'erreur

Retour sur l'écriture de javadoc: Quelques balises supplémentaires

- **@see** *NomClasse#methode(arguments)*

  renvoi à la documentation d'une autre méthode ou classe.
  Exemple:

  @see MiseEnPage#justifierGauche(String, int)

- **@author** *NOM*

  - nom du créateur de la classe ou de la méthode

- {**@link** *NomClasse#methode(arguments)*} comme @see, mais @see donne un lien après la documentation, alors que @link se place dans le corps de la documentation.

Intégration de javadoc dans eclipse

Aide lors de la complétion

```
ıblic class MiseEnPage {
   public static void justifierGauche(String l, int largeur) {
       System.out.println
   }

   public static void jus
       afficherMarge(larg
       System.out.println
   }


public static double c
    return x*x;
}

public static void centrer(String l, int largeur) {
```

- println() : void – PrintStream
- println(boolean x) : void – PrintStream
- println(char x) : void – PrintStream
- println(char[] x) : void – PrintStream
- println(double x) : void – PrintStream
- println(float x) : void – PrintStream
- println(int x) : void – PrintStream
- println(long x) : void – PrintStream
- println(Object x) : void – PrintStream
- println(String x) : void – PrintStream

Press '^Space' to show Template Proposals

println

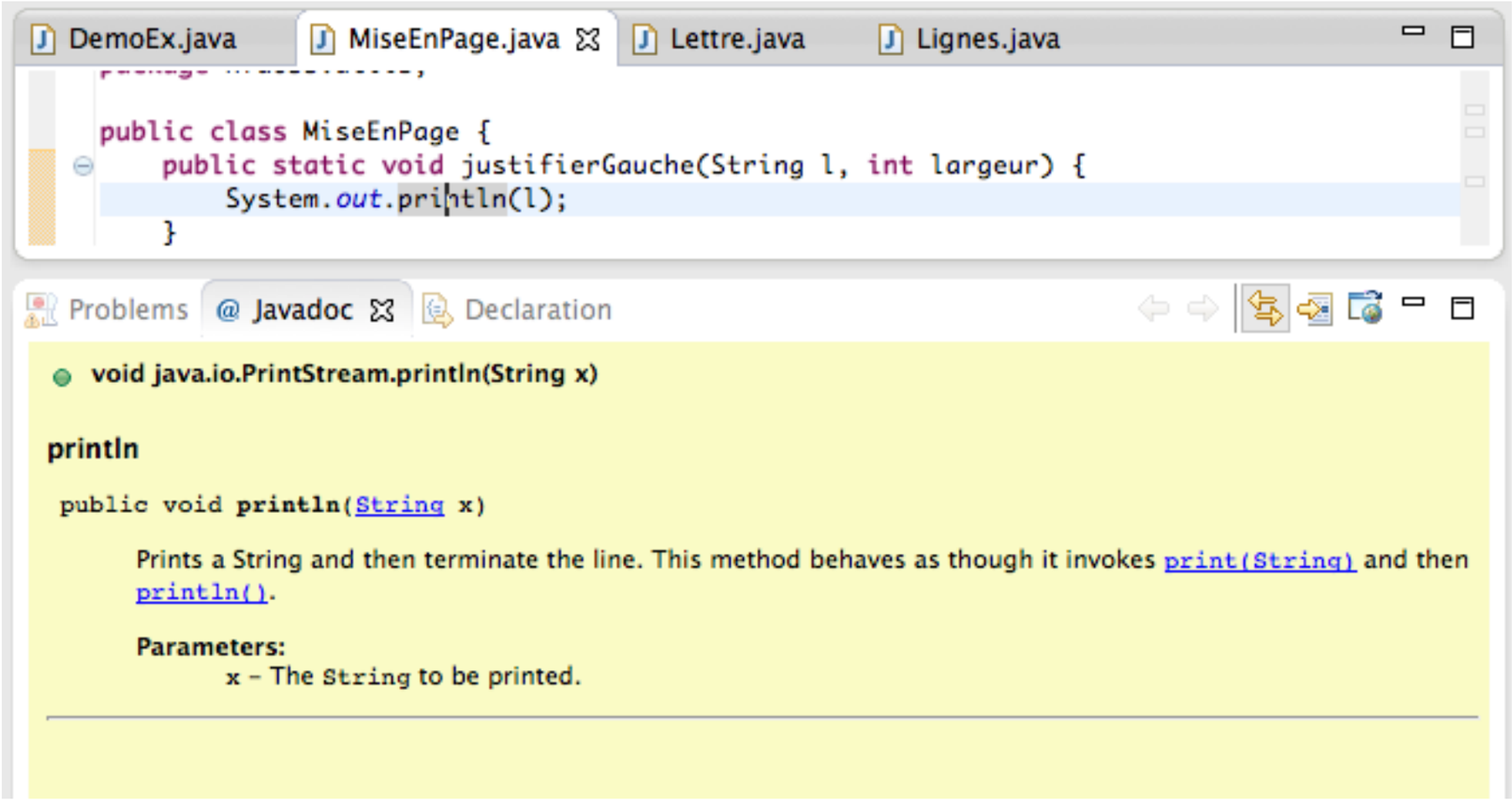
`public void` **`println`**`(int x)`

Prints an integer and then terminate the line. This method behaves as though it invokes `print(int)` and then `println()`.

Parameters:
`x` – The `int` to be printed.

Press 'Tab' from proposal table or click for focus

Onglet javadoc



Aide à l’écriture

```
/**
public static void justifierDroit(String l, int largeur) {
        afficherMarge(largeur- l.length());
        System.out.println(l);
}
```

on tape « entrée » et…

```
/**
 *
 * @param l
 * @param largeur
 */
public static void justifierDroit(String l, int largeur) {
    afficherMarge(largeur- l.length());
    System.out.println(l);
}
```

Tests et JUnit 4

Tests

- But : détecter 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, outil extrêmement important pour développer des logiciels robustes
- automatisés avec JUNIT

Tests unitaires / test fonctionnels

- **test unitaire**: on teste qu'un sous programme fonctionne correctement
  - c'est à dire que si on l'appelle en respectant les préconditions, les postconditions seront vérifiées
- **test fonctionnel**: on teste un scénario, composé de plusieurs appels de sous programmes

Test de non régression

- **non régression**: on conserve les tests, et on les rejoue quand le logiciel évolue.
  - on détecte ainsi 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 regressions

Test-driven development (très simplifié)

- ***TDD*** ou ***développement piloté par les tests***
- On commence par définir les interfaces des sous-programme (leur en-tête)
- Ensuite on écrit les tests que ces sous programmes doivent satisfaire
- Ensuite seulement, on écrit le code des sous programme
- tester, recommencer jusqu'à ce que tous les tests passent

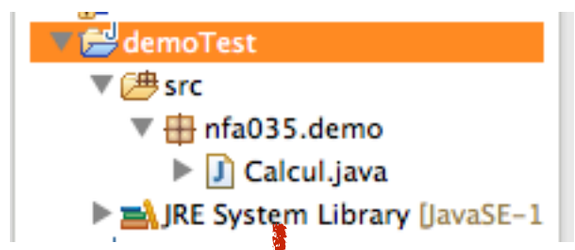
Notre classe à tester…

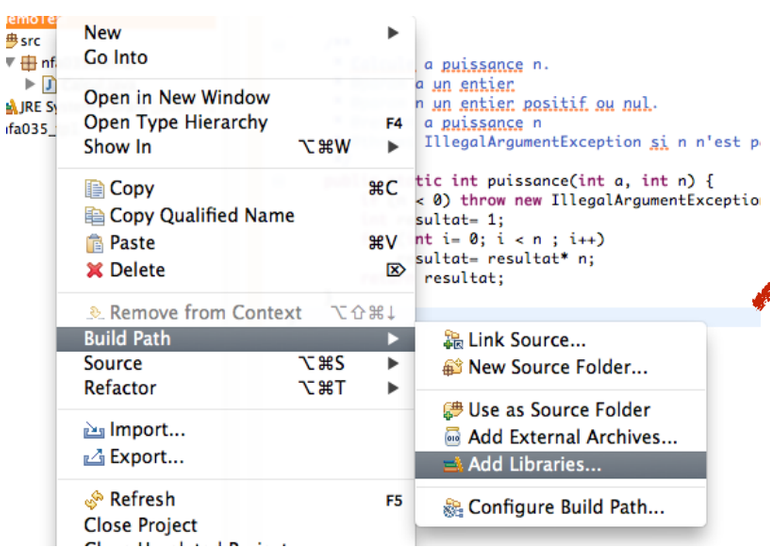
```
package nfa035.demo;

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;
  }
}
```

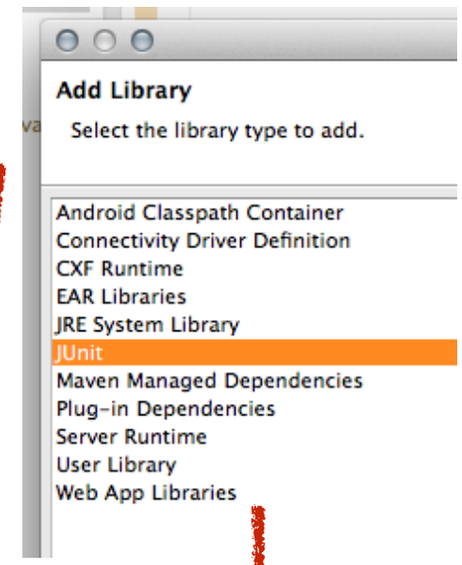
(attention, buggée!!!)

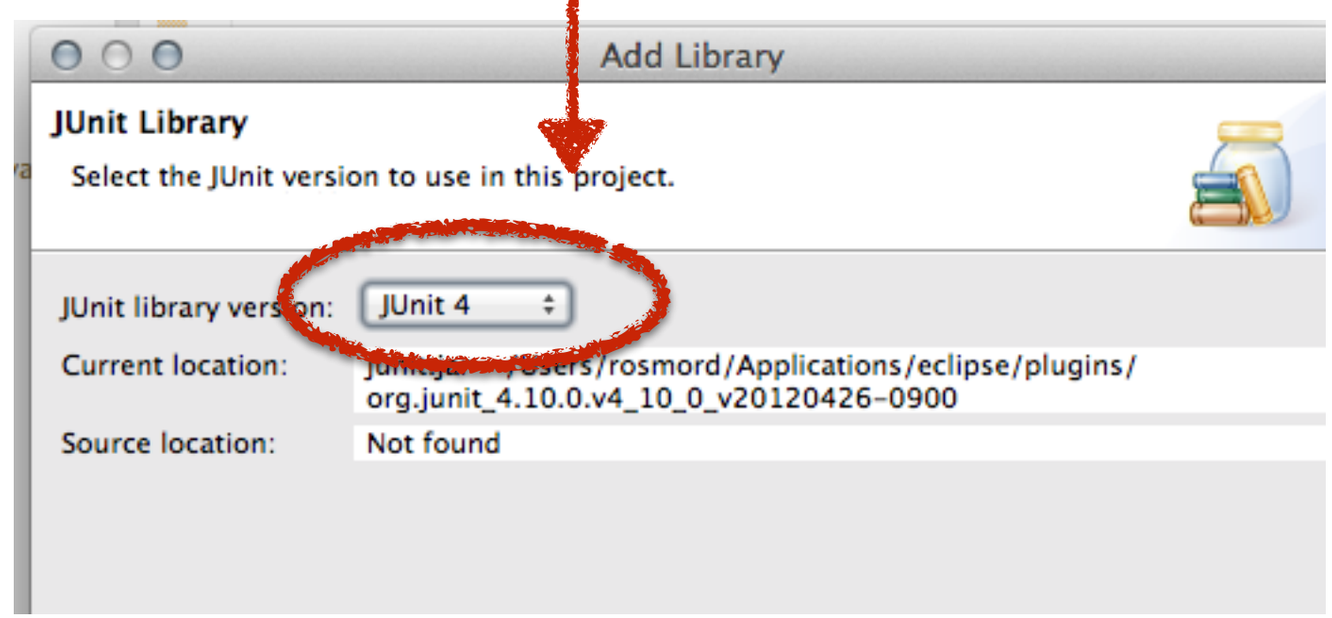
JUnit en pratique

- Ajouter JUnit4 à son projet



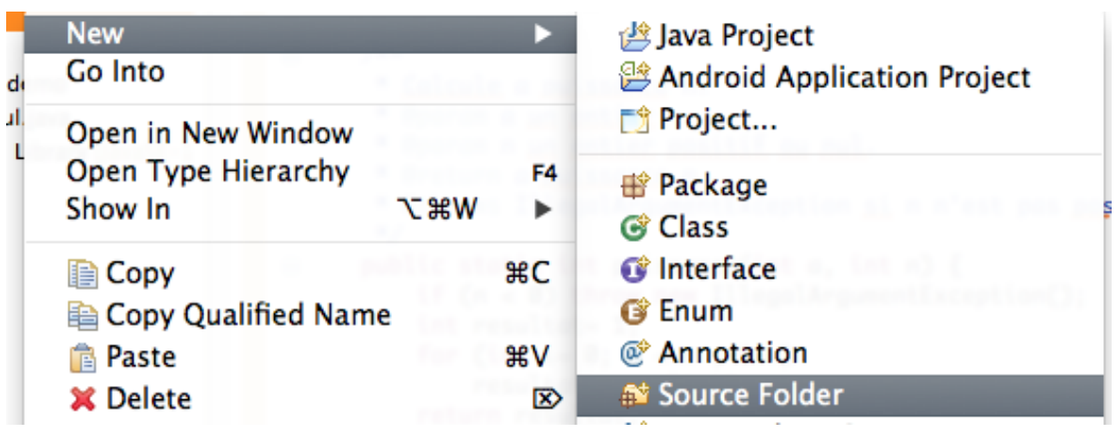


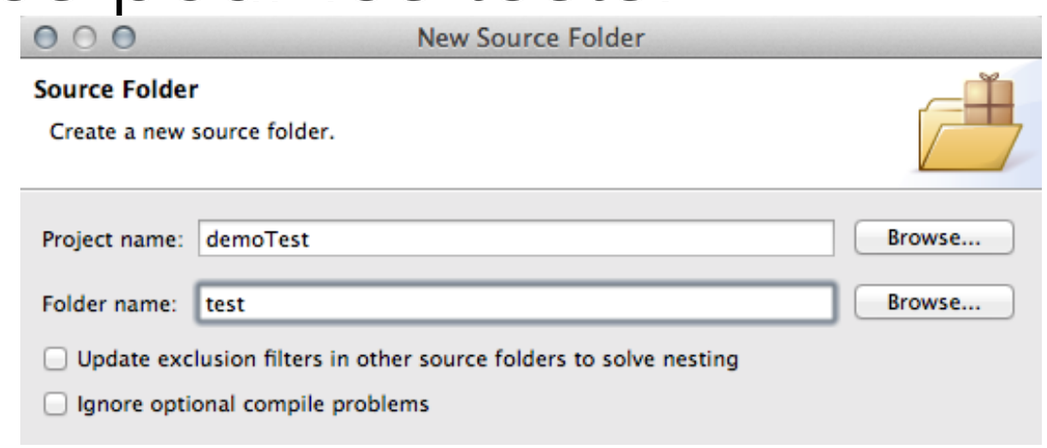


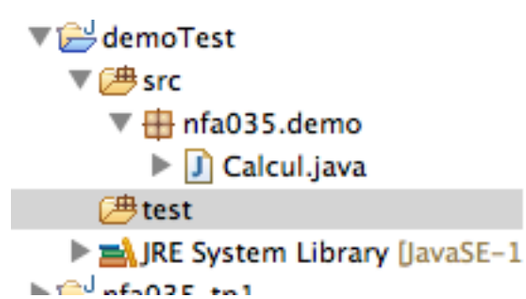


(optionnel)

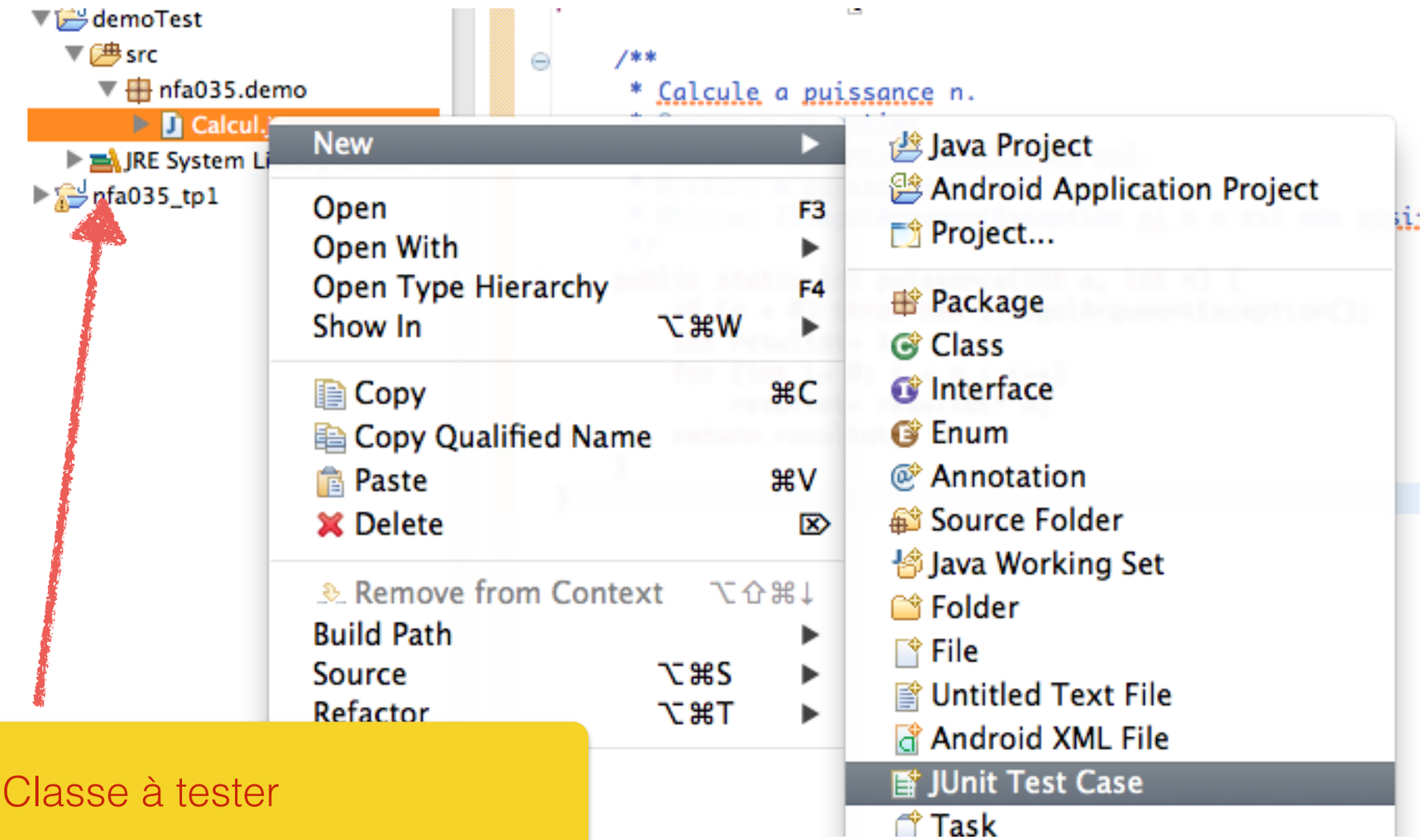
- Créer un dossier sources pour les tests:





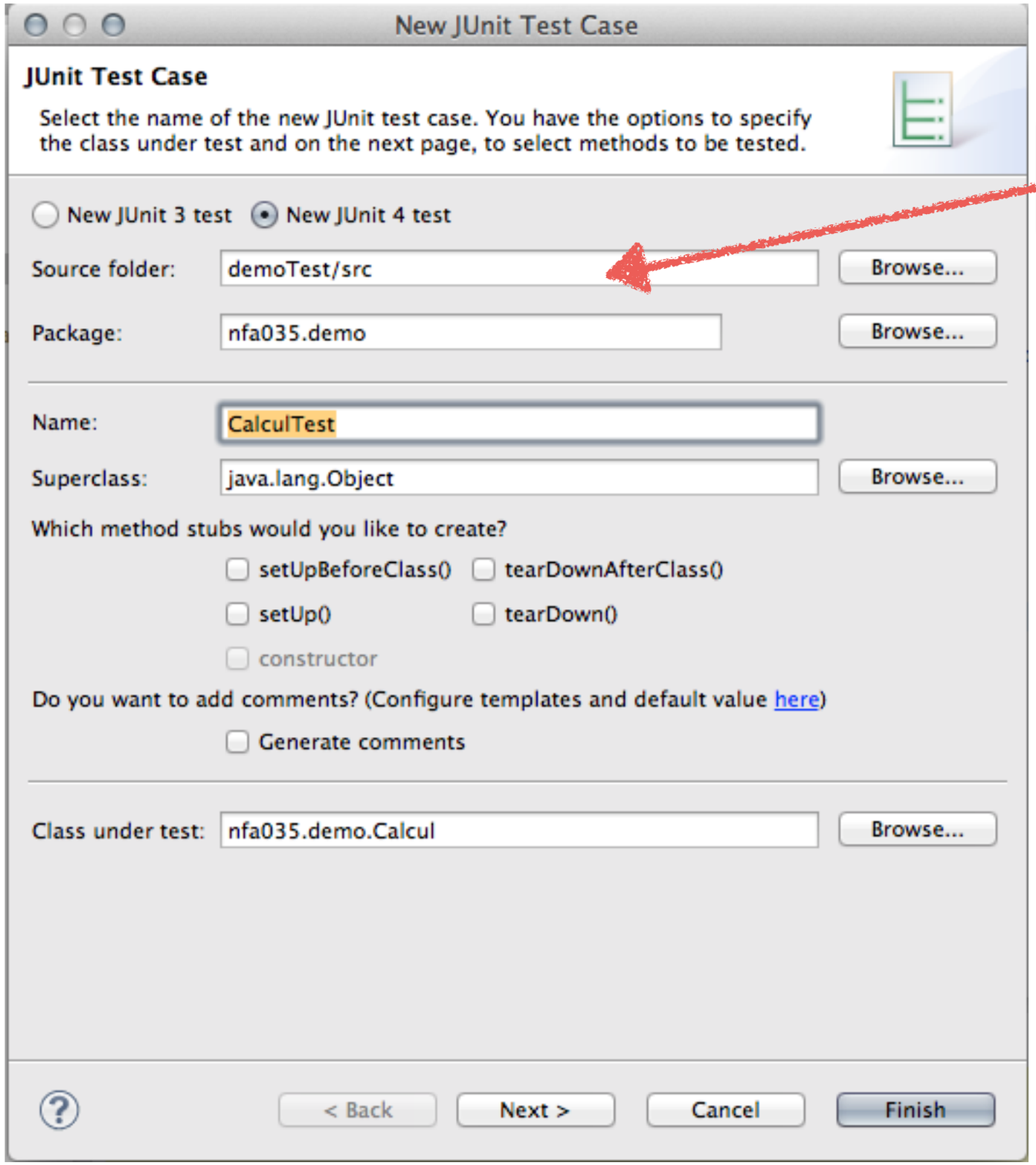


Création du test



Classe à tester

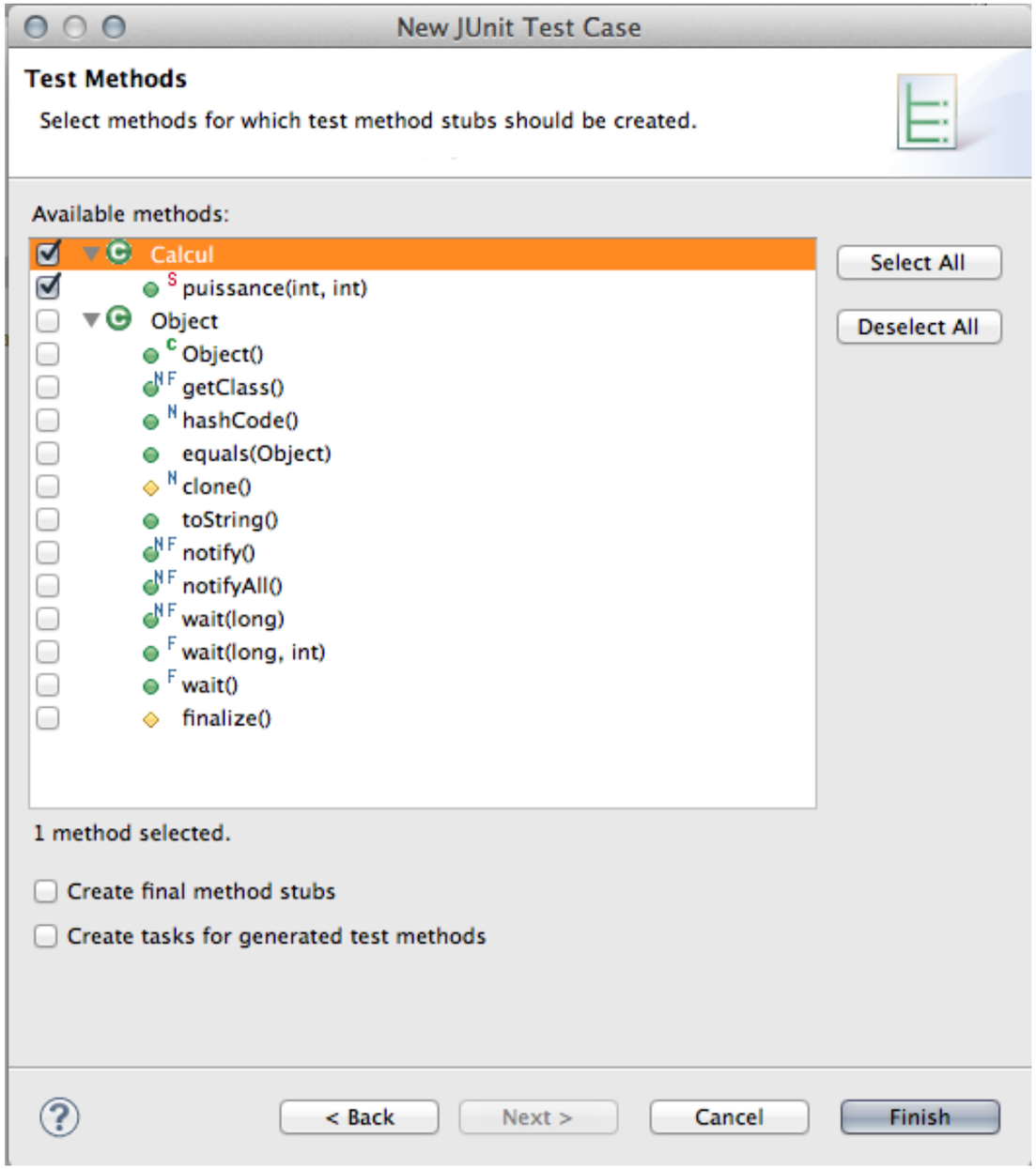
Création du test



Modifier éventuellement le dossier

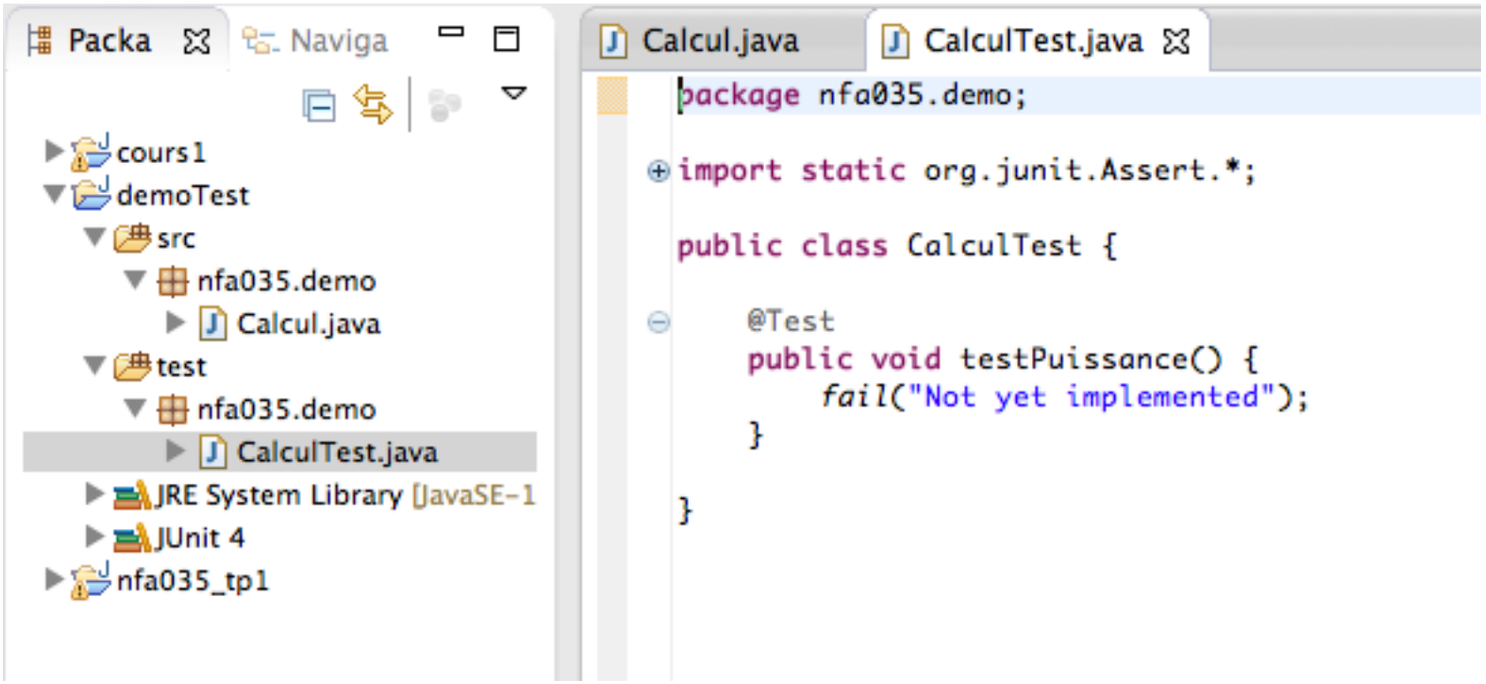
Création du test

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



Création du test

- Le test est normalement dans le même package que la classe à tester



Écriture des tests

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

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

- Une méthode de test est précédée de l'annotation **@Test**

- Dans la méthode de test, on appelle la méthode à tester, et on vérifie que le résultat est correct avec les méthodes de la classe Assert

- On écrit une méthode par test à effectuer..

Écriture de tests

```
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));
  }
```

Annotation test

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

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 exception

- 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 :

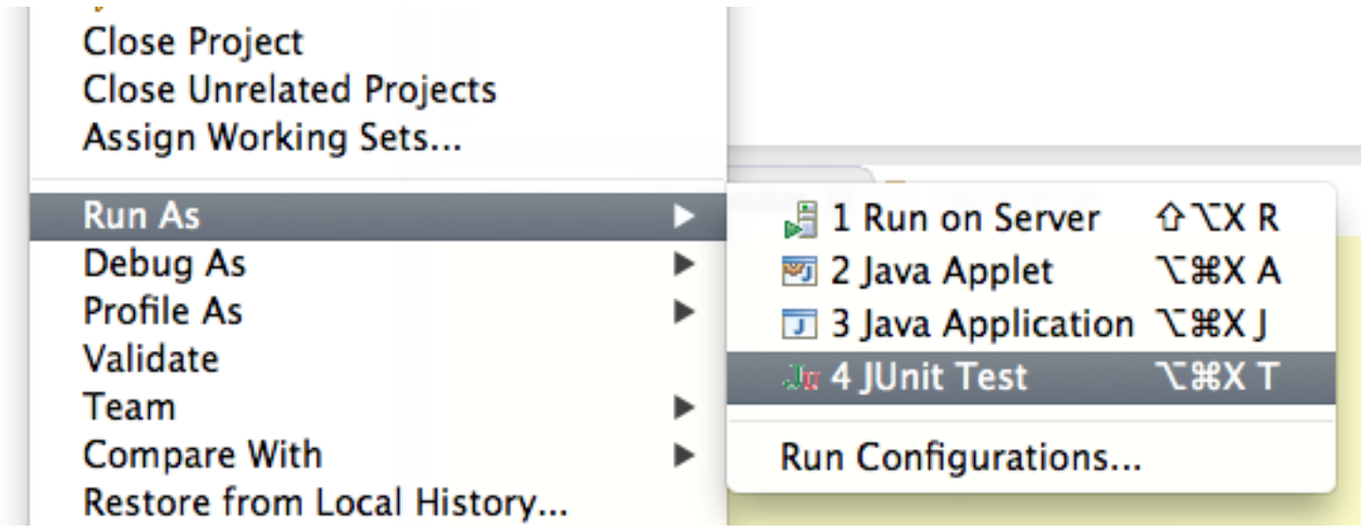
@Test(timeout=1000)

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

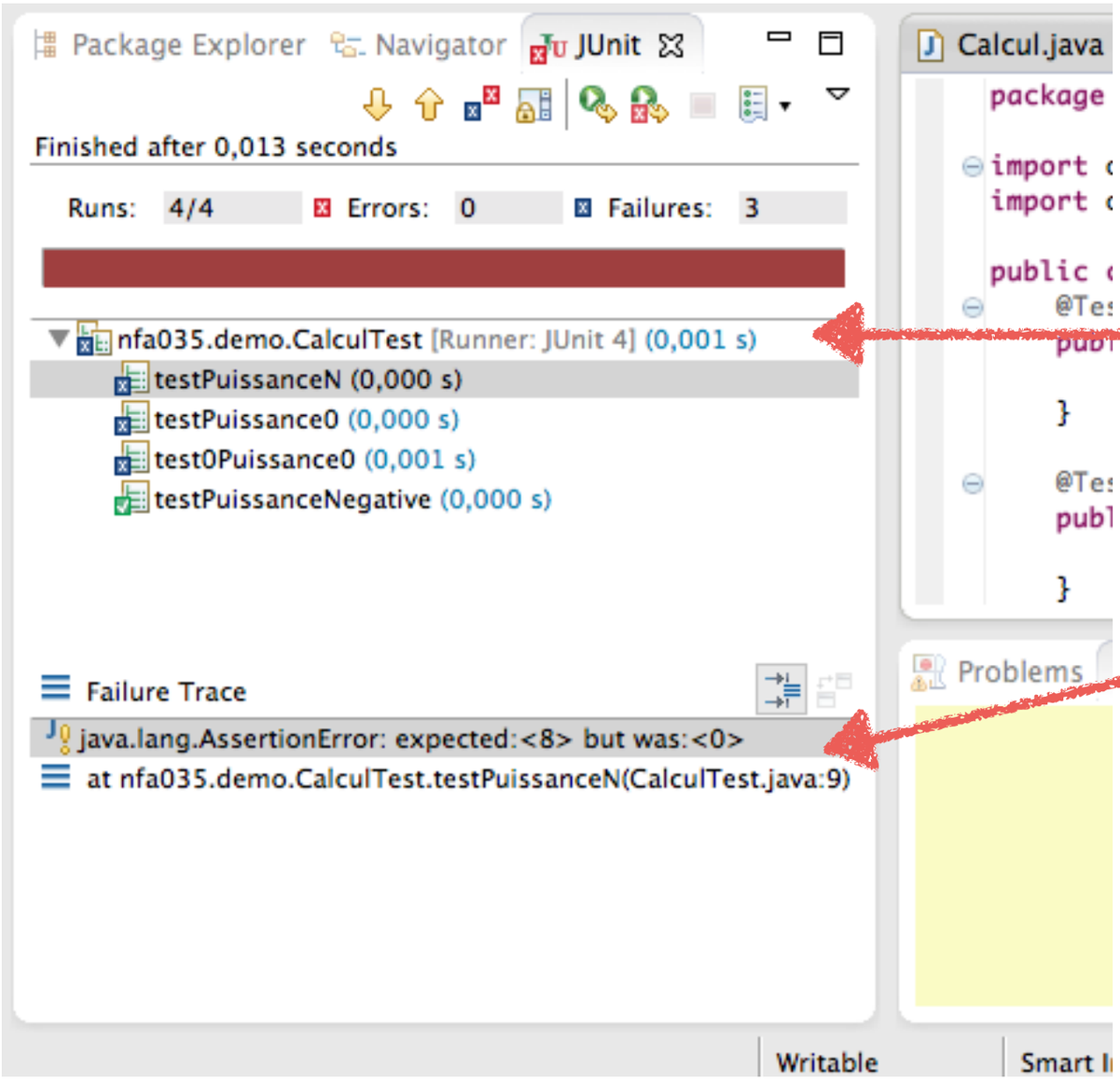
Les tests pour notre classe

```
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



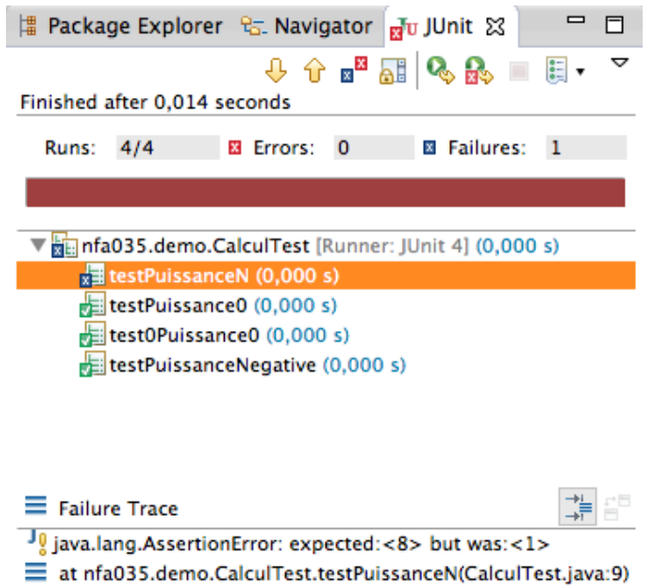
Faire tourner les tests dans eclipse



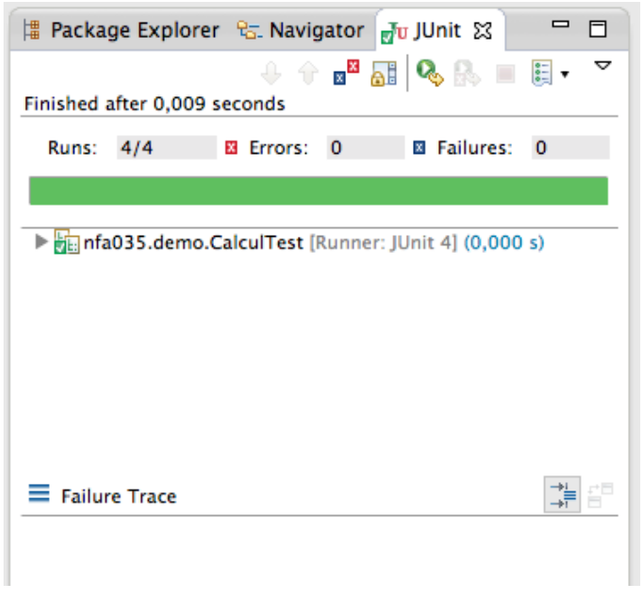
- 4 tests, 3 échecs…
- sur le premier : on attendait « 8 », mais le résultat fut 3

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 :

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

```
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);
```

```
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.)*