Les Framework Java

JUnit

Claude Duvallet

Université du Havre
UFR Sciences et Techniques
25 rue Philippe Lebon - BP 540
76058 LE HAVRE CEDEX
Claude.Duvallet@gmail.com
http://litis.univ-lehavre.fr/~duvallet/

JUnit

Présentation (1/4)

- JUnit est une bibliothèque de tests unitaires pour Java.
- Créé par Kent Beck et Erich Gamma, JUnit est certainement le projet de la série des xUnit connaissant le plus de succès.
- JUnit définit deux types de fichiers de tests :
  - Les `TestCase` sont des classes contenant un certain nombre de méthodes de tests. Un TestCase sert généralement à tester le bon fonctionnement d'une classe.
  - Une `TestSuite` permet d'exécuter un certain nombre de `TestCase` déjà définis.
- Junit est intégré par défaut dans les environnements de développement Java tels qu'Eclipse et Netbeans.
- Le principal intérêt de JUnit est de s'assurer que le code répond toujours aux besoins même après d'éventuelles modifications.
- Plus généralement, ce type de tests est appelé tests unitaires de non régression.

Présentation (2/4)

- JUnit propose :
  - Un framework pour le développement des tests unitaires reposant sur des assertions qui testent les résultats attendus.
  - Des applications pour permettre l'exécution des tests et afficher les résultats.
- Le but est d'automatiser les tests qui sont exprimés dans des classes sous la forme de cas de tests avec leurs résultats attendus.
- JUnit exécute ces tests et les comparent avec ces résultats.
- Cela permet de séparer le code de la classe, du code qui permet de la tester.
- Souvent pour tester une classe, il est facile de créer une méthode `main()` qui va contenir les traitements de tests.
- L'inconvénient est que ce code "superflu" aux traitements proprement dit est qu'il soit inclus dans la classe.

Présentation (3/4)

- De plus, son exécution doit se faire manuellement.
- La rédaction de cas de tests peut avoir un effet immédiat pour détecter des bugs mais surtout elle a un effet à long terme qui facilite la détection d'effets de bords lors de modifications.
- Les cas de tests sont regroupés dans des classes Java qui contiennent une ou plusieurs méthodes de tests.
- Ces cas de tests peuvent être regroupés sous la forme de suites de tests.
- Les cas de tests peuvent être exécutés individuellement ou sous la forme de suites de tests.
- JUnit permet le développement incrémental d'une suite de tests.
- Avec JUnit, l'unité de test est une classe dédiée qui regroupe des cas de tests.

Présentation (4/4)

- Ces cas de tests exécutent les tâches suivantes :
  - création d'une instance de la classe et de tout autre objet nécessaire aux tests
  - appel de la méthode à tester avec les paramètres du cas de test
  - comparaison du résultat attendu avec le résultat obtenu : en cas d'échec, une exception est levée
- JUnit est particulièrement adapté pour être utilisé avec la méthode eXtreme Programming (XP) puisque cette méthode préconise entre autre l'automatisation des tâches de tests unitaires qui ont été définis avant l'écriture du code.
- La page officielle est à l'url : `http://junit.org`.
- La dernière version de JUnit peut être téléchargée sur le site `www.junit.org`.
- Pour pouvoir utiliser JUnit, il faut ajouter le fichier `junit.jar` au classpath.

Un premier exemple (1/3)

- Soit la classe suivante :

```
public class MaClasse{
    public static int calculer(int a, int b) {
        int res = a + b;
        if (a == 0){
            res = b * 2;
        }

        if (b == 0) {
            res = a * a;
        }
        return res;
    }
}
```

  - Il faut compiler cette classe : `javac MaClasse.java`
  - Il faut ensuite écrire une classe qui va contenir les différents tests à réaliser par JUnit.

Un premier exemple (2/3)

- Voici la classe de test :

```
import junit.framework.*;

public class MaClasseTest extends TestCase{

    public void testCalculer() throws Exception {
        assertEquals(2,MaClasse.calculer(1,1));
    }
}
```

  - Il faut compiler cette classe avec le fichier `junit.jar` qui doit être dans le classpath.
  - Enfin, il suffit d'appeler JUnit pour qu'il exécute la séquence de tests.
- Voici le résultat de l'exécution :

```
javac -cp junit.jar;. MaClasseTest.java
java -cp junit.jar;. junit.textui.TestRunner MaClasseTest
.
Time: 0.005

OK (1 test)
```

Un premier exemple (3/3)

- Attention : le respect de la case dans le nommage des méthodes de tests est très important.
  - Les méthodes de tests doivent obligatoirement commencer par test en minuscule car JUnit utilise l'introspection pour déterminer les méthodes à exécuter.
  - Exemple :

```
import junit.framework.*;

public class MaClasseTest extends TestCase{

    public void TestCalculer() throws Exception {
        assertEquals(2,MaClasse.calculer(1,1));
    }
}
```

  - L'utilisation de cette classe avec JUnit produit le résultat suivant :

```
java -cp junit.jar;. junit.textui.TestRunner MaClasseTest
.F
Time: 0.001
There was 1 failure:
1) warning(junit.framework.TestSuite$1)junit.framework.AssertionFailedError:
                                        No tests found in MaClasseTest

FAILURES!!!
Tests run: 1,  Failures: 1,  Errors: 0
```

L'écriture des cas de tests (1/2)

- JUnit propose un framework pour écrire les classes de tests.
- Un test est une classe qui hérite de la classe TestCase.
- Par convention le nom de la classe de test est composé du nom de la classe suivi de Test.
- Chaque cas de tests fait l'objet d'une méthode dans la classe de tests.
- Le nom de ces méthodes doit obligatoirement commencer par le suffixe test.

L'écriture des cas de tests (2/2)

- Chacune de ces méthodes contient généralement des traitements en trois étapes :
  - Instanciation des objets requis
  - Invocation des traitements sur les objets
  - Vérification des résultats des traitements
- Il est important de se souvenir lors de l'écriture de cas de tests que ceux-ci doivent être indépendants les uns des autres.
- JUnit ne garantit pas l'ordre d'exécution des cas de tests puisque ceux-ci sont obtenus par introspection.
- Toutes les classes de tests avec JUnit héritent de la classe Assert.

La définition de la classe de tests (1/3)

- Pour écrire les cas de tests, il faut écrire une classe qui étende la classe `junit.framework.TestCase`.
- Le nom de cette classe est le nom de la classe à tester suivi par "Test".
- Une classe de tests doit obligatoirement posséder un constructeur qui attend un objet de type String en paramètre.
- Exemple :

```
import junit.framework.*;

public class MaClasseTest extends TestCase{
    public MaClasseTest(String testMethodName) {
        super(testMethodName);
    }

    public void testCalculer() throws Exception {
        fail("Cas de test a ecrire");
    }
}
```

La définition de la classe de tests (2/3)

- Dans la classe précédente, il faut écrire une méthode dont le nom commence par "test" en minuscule suivi du nom du cas de test (généralement le nom de la méthode à tester).
- Chacune de ces méthodes doit avoir les caractéristiques suivantes :
  - elle doit être déclarée public
  - elle ne doit renvoyer aucune valeur
  - elle ne doit pas posséder de paramètres.
- Par introspection, JUnit va automatiquement rechercher toutes les méthodes qui respectent cette convention.
- Le respect de ces règles est donc important pour une bonne exécution des tests par JUnit.

La définition de la classe de tests (3/3)

- Exemple : la méthode commence par T et non t

```
import junit.framework.*;

public class MaClasseTest extends TestCase{

    public void TestCalculer() throws Exception {
        // assertEquals(2,MaClasse.calculer(1,1));
        fail("Cas de test a ecrire");
    }
}
```

- Résultat :

```
java -cp junit.jar;. junit.textui.TestRunner MaClasseTest
.F
Time: 0,01
There was 1 failure:
1) warning(junit.framework.TestSuite$1)junit.framework.AssertionFailedError: No
tests found in MaClasseTest

FAILURES!!!
Tests run: 1,  Failures: 1,  Errors: 0
```

La définition des cas de tests (1/6)

- Chaque classe de tests doit avoir obligatoirement au moins une méthode de test sinon une erreur est remontée par JUnit.
- La découverte des méthodes de tests par Junit repose sur l'introspection :
  - JUnit recherche les méthodes qui débutent par test, n'ont aucun paramètre et ne retourne aucune valeur.
  - Ces méthodes peuvent lever des exceptions qui sont automatiquement capturées par JUnit qui remonte alors une erreur et donc un échec du cas de tests.
- Dès qu'un test échoue, l'exécution de la méthode correspondante est interrompue et JUnit passe à méthode suivante.
- Avec JUnit, la plus petite unité de tests est l'assertion dont le résultat de l'expression booléenne indique un succès ou une erreur.

La définition des cas de tests (2/6)

- Les cas de tests utilisent des affirmations (assertion en anglais) sous la forme de méthodes nommées `assertXXX()`.
- Il existe de nombreuses méthodes de ce type qui sont héritées de la classe `junit.framework.Assert` :

Méthode	Rôle
assertEquals()	Vérifier l'égalité de deux valeurs de type primitif ou objet (en utilisant la méthode equals()). Il existe de nombreuses surcharges de cette méthode pour chaque type primitif, pour un objet de type Object et pour un objet de type String.
assertFalse()	Vérifier que la valeur fournie en paramètre est fausse.
assertNull()	Vérifier que l'objet fourni en paramètre soit null.
assertNotNull()	Vérifier que l'objet fourni en paramètre ne soit pas null.
assertSame()	Vérifier que les deux objets fournis en paramètre font référence à la même entité. Exemples identiques : assertSame("Les deux objets sont identiques", obj1, obj2) ; assertTrue("Les deux objets sont identiques ", obj1 == obj2) ;
assertNotSame()	Vérifier que les deux objets fournis en paramètre ne font pas référence à la même entité.
assertTrue()	Vérifier que la valeur fournie en paramètre est vraie.

La définition des cas de tests (3/6)

- Bien qu'il serait possible de n'utiliser que la méthode assertTrue(), les autres méthodes assertXXX() facilite l'expression des conditions de tests.
- Chacune de ces méthodes possède une version surchargée qui accepte un paramètre supplémentaire sous la forme d'une chaîne de caractères indiquant un message qui sera affiché en cas d'échec du cas de test.
- Le message doit décrire le cas de test évalué à "true".
- L'utilisation de cette version surchargée est recommandée car elle facilite l'exploitation des résultats des cas de tests.
- Exemple :

```
import junit.framework.*;

public class MaClasse2Test extends TestCase{

    public void testCalculer() throws Exception {
        MaClasse2 mc = new MaClasse2(1,1);
        assertEquals(2,mc.calculer());
    }
}
```

La définition des cas de tests (4/6)

- L'ordre des paramètres contenant la valeur attendue et la valeur obtenue est important pour obtenir un message d'erreur fiable en cas d'échec du cas de test.
- Quelque soit la surcharge utilisée l'ordre des deux valeurs à tester est toujours la même : c'est toujours la valeur attendue qui précède la valeur courante.
- La méthode `fail()` permet de forcer le cas de test à échouer.
- Une version surchargée permet de préciser un message qui doit être afficher.
- Il est aussi souvent utile lors de la définition des cas de tests de devoir tester si une exception est levée lors de l'exécution.

La définition des cas de tests (5/6)

- Exemple :

```
import junit.framework.*;

public class MaClasse2Test extends TestCase{
    public void testSommer() throws Exception {
        MaClasse2 mc = new MaClasse2(0,0);
        mc.sommer();
    }
}
```

- Résultat :

```
java -cp junit.jar;. junit.textui.TestRunner MaClasse2Test
.E
Time: 0,01
There was 1 error:
1) testSommer(MaClasse2Test)java.lang.IllegalStateException:
                                        Les deux valeurs sont nulles
        at MaClasse2.sommer(MaClasse2.java:42)
        at MaClasse2Test.testSommer(MaClasse2Test.java:31)
        at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
        at sun.reflect.NativeMethodAccessorImpl.invoke(
                                         NativeMethodAccessorImpl.java:39)
        at sun.reflect.DelegatingMethodAccessorImpl.invoke(
                                    DelegatingMethodAcces.sorImpl.java:25)

FAILURES!!!
Tests run: 2,  Failures: 0,  Errors: 1
```

La définition des cas de tests (6/6)

- Avec JUnit, pour réaliser de tels cas de tests, il suffit d'appeler la méthode avec les conditions qui doivent lever une exception, d'encapsuler cet appel dans un bloc try/catch et d'appeler la méthode fail() si l'exception désirée n'est pas levée.
- Exemple :

```
import junit.framework.*;

public class MaClasse2Test extends TestCase{

    public void testSommer() throws Exception {
        MaClasse2 mc = new MaClasse2(1,1);

        // cas de test 1
        assertEquals(2,mc.sommer());

        // cas de test 2
        try {
            mc.setA(0);
            mc.setB(0);
            mc.sommer();
            fail("Une exception de type IllegalStateException aurait
                                                    du etre levee");
        } catch (IllegalStateException ise) {}
    }
}
```

L'initialisation des cas de tests

- Il est fréquent que les cas de tests utilisent plusieurs instances d'un même objet ou nécessitent l'usage de ressources particulières telles qu'une instance d'une classe pour l'accès à une base de données par exemple.
- Pour réaliser ces opérations de création et de destruction d'objets, la classe TestCase propose les méthodes `setUp()` et `tearDown()` qui sont respectivement appelées avant et après l'appel de chaque méthode contenant un cas de test.
- Il suffit simplement de les redéfinir en fonction des besoins.
- Le plus simple est de définir un membre privé du type dont on a besoin et de créer une instance de ce type dans la méthode `setUp()`.
- Il est important de se souvenir que `setUp()` est invoquée systématiquement avant l'appel de chaque méthode de tests.

L'initialisation des cas de tests

- Sa mise en œuvre n'est donc requise que si toutes les méthodes de tests ont besoin de créer une instance d'un même type ou d'exécuter un même traitement.
- Ceci évite de créer l'instance dans chaque méthode de tests et simplifie donc l'écriture des cas de tests.
- La méthode `tearDown()` peut avoir un grand intérêt pour par exemple libérer des ressources comme une connexion à une base de données initiée dans la méthode `setUp()`.
- Pour des besoins particuliers, il peut être nécessaire d'exécuter du code une seule fois avant l'exécution des cas de tests et/ou exécuter du code une fois tous les cas des tests exécutés.
- JUnit propose pour cela la classe `junit.Extensions.TestSetup` qui propose la mise en œuvre du design pattern décorateur.

Le test de la levée d'exceptions

- Il est fréquent qu'une méthode puisse lever une ou plusieurs exceptions durant son exécution.
- Il faut prévoir des cas de tests pour vérifier que dans les conditions adéquates une exception attendue est bien levée.
- Pour effectuer la vérification de la levée d'une exception, il faut inclure l'invocation de la méthode dans un bloc `try/catch` et faire appel à la méthode `fail()` si l'exception n'est pas levée.
- Attention : une erreur courante lorsque l'on code ces premiers tests unitaires est d'inclure les invocations de méthodes dans des blocs `try/catch`.
- Leur utilisation doit être uniquement réservée.
- Dans tous les autres cas, il faut laisser l'exception se propager : JUnit va automatiquement reporter un échec du test.
- Il est en particulier inutile d'utiliser un bloc `try/catch` et de faire appel à `fail()` dans le catch puisque JUnit le fait déjà.

L'héritage d'une classe de base

- Il est possible de définir une classe de base qui servira de classe mère à d'autres classes de tests notamment en leur fournissant des fonctionnalités communes.
- JUnit n'impose pas qu'une classe de tests dérive directement de la classe `TestCase`.
- Ceci est particulièrement pratique lorsque l'on souhaite que certaines initialisations ou certains traitements soit systématiquement exécutés (exemple chargement d'un fichier de configuration, ...).
- Il est par exemple possible de faire des initialisations dans le constructeur de la classe mère et invoquer ce constructeur dans les constructeurs des classe filles.

L'exécution des tests (1/2)

- JUnit propose trois applications différentes nommées `TestRunner` pour exécuter les tests en mode ligne de commande ou application graphique :
  - une application console : `junit.textui.TestRunner` qui est très rapide et adaptée à une intégration dans un processus de générations automatiques.
  - une application graphique avec une interface Swing : `junit.swingui.TestRunner`.
  - une application graphique avec une interface AWT : `junit.awtui.TestRunner`.
- Quelque soit l'application utilisée, les entités suivantes doivent être incluses dans le classpath :
  - le fichier `junit.jar`.
  - les classes à tester et les classes des cas de tests.
  - les classes et bibliothèques dont toutes ces classes dépendent.

L'exécution des tests (2/2)

- Suite à l'exécution d'un cas de test, celui ci peut avoir un des trois états suivants :
  - échoué : une exception de type `AssertionFailedError` est levée.
  - en erreur : une exception non émise par le framework et non capturée a été levée dans les traitements.
  - passé avec succès.
- L'échec d'un seul cas de test entraîne l'échec du test complet.
- L'échec d'un cas de test peut avoir plusieurs origines :
  - le cas de test contient un ou plusieurs bugs.
  - le code à tester contient un ou plusieurs bugs.
  - le cas de test est mal défini.
  - une combinaison des cas précédents simultanément.

L'exécution des tests dans la console (1/2)

- L'utilisation de l'application console nécessite quelques paramètres lors de son utilisation :

```
java -cp junit.jar;. junit.textui.TestRunner MaClasseTest
```

- Le seul paramètre obligatoire est le nom de la classe de tests.
- Celle ci doit obligatoirement être sous la forme pleinement qualifiée si elle appartient à un package.
- Il est possible de faire appel au `TestRunner` dans une application en utilisant sa méthode `run()` et en lui passant en paramètre un objet de type `Class` qui encapsule la classe de tests à exécuter.
- Le `TestRunner` affiche le résultat de l'exécution des tests dans la console.
- La première ligne contient un caractère point pour chaque test exécuté.

L'exécution des tests dans la console (2/2)

- Lorsque de nombreux tests sont exécutés cela permet de suivre la progression.
- Le temps total d'exécution en seconde est ensuite affiché sur la ligne "Time :"
- En cas d'erreur, la première ligne contient un F à la suite du caractère point correspondant au cas de test en échec.
- Le résumé de l'exécution affiche le détail de chaque cas de tests qui a échoué.
- Les cas en échec (failures) correspondent à une vérification faite par une méthode assertXXX() qui a échoué.
- Les cas en erreur (errors) correspondent à la levée inattendue d'une exception lors de l'exécution du cas de test.

L'exécution des tests dans une application graphique (1/2)

- Pour utiliser des classes de tests avec ces applications graphiques, il faut obligatoirement que les classes de tests et toutes celles dont elles dépendent soient incluses dans le CLASSPATH.
- Elles doivent obligatoirement être sous la forme de fichier .class non incluses dans un fichier jar.

```
java -cp junit.jar;. junit.swingui.TestRunner MaClasseTest
```

- Il suffit de cliquer sur le bouton "Run" pour lancer l'exécution des tests.
- La case à cocher "Reload classes every run" indique à JUnit de recharger les classes à chaque exécution.
- Ceci est très pratique car cela permet de modifier les classes et de laisser l'application de tests ouverte.

L'exécution des tests dans une application graphique (2/2)

- Si un ou plusieurs tests échouent la barre de résultats n'est plus verte mais rouge.
- Dans ce cas, le nombre d'erreurs et d'échecs est affiché ainsi que leur liste complète.
- Il suffit d'en sélectionner un pour obtenir le détail de la raison du problème.
- Il est aussi possible de ne réexécuter que le cas sélectionné.

L'exécution répétée d'un cas de test

- JUnit propose la classe `junit.extensions.RepeatedTest` qui permet d'exécuter plusieurs fois la même suite de tests.
- Le constructeur de cette classe attend en paramètre une instance de la suite de tests et le nombre de répétitions de l'exécution de la suite de tests.
- Exemple :

```
import junit.extensions.RepeatedTest;
import junit.framework.*;

public class PersonneTest extends TestCase {

  private Personne personne;

  public PersonneTest(String name) {
    super(name);
  }

  public static Test suite() {
    return new RepeatedTest(new TestSuite(PersonneTest.class), 5);
  }

  public static void main(String[] args) {
      junit.textui.TestRunner.run(suite());
  }
}
```

L'exécution concurrente de tests

- JUnit propose la classe `junit.extensions.ActiveTestSuite` qui permet d'exécuter plusieurs suites de tests chacune dans un thread dédié.
- Ainsi l'exécution des suites de tests se fait de façon concurrente.
- L'ensemble de la suite de tests ne se termine que lorsque tous les threads sont terminés.
- Même si cela n'est pas recommandé, la classe `ActiveTestSuite` peut être utilisée comme un outil de charge rudimentaire.
- Il est possible de combiner l'utilisation des classes `ActiveTestsuite` et `RepeatedTest`.

Les suites de tests (1/2)

- Les suites de tests permettent de regrouper plusieurs tests dans une même classe.
- Ceci permet l'automatisation de l'ensemble des tests inclus dans la suite et de préciser leur ordre d'exécution.
- Pour créer une suite, il suffit de créer une classe de type `TestSuite` et d'appeler la méthode `addTest()` pour chaque classe de tests à ajouter.
- Celle ci attend en paramètre une instance de la classe de tests qui sera ajoutée à la suite.
- L'objet de type TestSuite ainsi créé doit être renvoyé par une méthode dont la signature doit obligatoirement être `public static Test suite()`.

Les suites de tests (2/2)

- Celle ci sera appelée par introspection par le TestRunner.
- Il peut être pratique de définir une méthode `main()` dans la classe qui encapsule la suite de tests pour pouvoir exécuter le `TestRunner` de la console en exécutant directement la méthode statique `Run()`.
- Ceci évite de lancer JUnit sur la ligne de commandes.
- Deux versions surchargées des constructeurs permettent de donner un nom à la suite de tests.
- Un constructeur de la classe `TestSuite` permet de créer automatiquement par introspection une suite de tests contenant tous les tests de la classe fournie en paramètre.
- La méthode addTestSuite() permet d'ajouter à une suite une autre suite.

JUnit 4

- JUnit version 4 est une évolution majeure depuis les quelques années d'utilisation de la version 3.8.
- Un des grands bénéfices de cette version est l'utilisation des annotations introduites dans Java 5.
- La définition des cas de tests et des tests ne se fait donc plus sur des conventions de nommage et sur l'introspection mais sur l'utilisation d'annotations ce qui facilite la rédaction des cas de tests.
- Une compatibilité descendante est assurée avec les suites de tests de JUnit 3.8.
- Le nom du package des classes de JUnit est différent entre la version 3 et 4 :
  - les classes de Junit 3 sont dans le package junit.framework.
  - les classes de Junit 4 sont dans le package org.junit.

La définition d'une classe de tests

- Une classe de tests n'a plus l'obligation d'étendre la classe TestCase sous réserve d'utiliser les annotations définies par JUnit et d'utiliser des imports static sur les méthodes de la classe org.junit.Assert.
- Exemple :

```
import org.junit.*;
import static org.junit.Assert.*;

public class MaClasse {

}
```

La définition des cas de tests (1/2)

- Les méthodes contenant les cas de tests n'ont plus d'obligation à utiliser la convention de nommage qui imposait de préfixer le nom des méthodes par test.
- Avec JUnit 4, il suffit d'annoter la méthode avec l'annotation `@Test`.
- Il est ainsi possible d'utiliser n'importe quelle méthode comme cas de test simplement en utilisant l'annotation `@Test`.
- Exemple :

```
@Test
public void getNom() {
  assertEquals("est ce que nom est correct", "nom1", personne.getNom());
}
```

- Ceci permet d'utiliser le nom de méthode que l'on souhaite.
- Il est cependant conseillé de définir et d'utiliser une convention de nommage qui facilitera l'identification des classes de tests et des cas de tests.

La définition des cas de tests (2/2)

- Il est par exemple possible de maintenir les conventions de nommage de JUnit 3.
- L'annotation `@Ignore` permet de demander au framework d'ignorer un cas de tests.
- Les cas de tests dans ce cas sont marqués avec la lettre I lors de leur exécution en mode console.
- Son utilisation ne doit pas devenir une solution à certains problèmes.
- JUnit 4 inclut deux nouvelles surcharges de la méthode `assertEquals()` qui permettent de comparer deux tableaux d'objets.
- La comparaison se fait sur le nombre d'occurrences dans les tableaux et sur l'égalité de chaque objet d'un tableau dans l'autre tableau.

L'initialisation des cas des tests (1/3)

- JUnit 3 imposait une redéfinition des méthodes `setUp()` et `tearDown()` pour définir des traitements exécutés systématiquement avant et après chaque cas de test.
- JUnit 4 propose simplement d'annoter la méthode exécutée avant avec l'annotation `@Before` et la méthode exécutée après avec l'annotation `@After`.
- Exemple :

```
@Before
public void initialiser() throws Exception {
  personne = new Personne("nom1","prenom1");
}

@After
public void nettoyer() throws Exception {
  personne = null;
}
```

L'initialisation des cas des tests (2/3)

- Il est possible d'annoter une ou plusieurs méthodes avec `@Before` ou `@After`.
- Dans ce cas, toutes les méthodes seront invoquées au moment correspondant à leur annotation.
- Il n'est pas nécessaire d'invoquer explicitement les méthodes annotées avec `@Before` et `@After` d'une classe mère.
- Tant que ces méthodes ne sont pas redéfinies, elles seront automatiquement invoquées lors de l'exécution des tests :
  - les méthodes annotées avec `@Before` de la classe mère seront invoquées avant celles de la classe fille.
  - les méthodes annotées avec `@After` de la classe fille seront invoquées avant celles de la classe mère.

L'initialisation des cas des tests (3/3)

- JUnit 4 propose simplement d'annoter une ou plusieurs méthodes exécutées avant l'exécution du premier cas de tests avec l'annotation `@BeforeClass` et une ou plusieurs méthodes exécutées après l'exécution de tous les cas de test de la classe avec l'annotation `@AfterClass`.
- Ces initialisations peuvent être très utiles notamment pour des connexions coûteuses à des ressources qu'il est préférable de ne réaliser qu'une seule fois plutôt qu'à chaque cas de tests.
- Ceci peut contribuer à améliorer les performances lors de l'exécution des tests.

Le test de la levée d'exceptions (1/2)

- Avec JUnit 3, pour vérifier la levée d'une exception dans un cas de test, il faut entourer l'appel du traitement dans un bloc `try/catch` et invoquer la méthode `fail()` à la fin du bloc `try`.
- JUnit 4 propose une annotation pour faciliter la vérification de la lever d'une exception.
- L'attribut `expected` de l'annotation `@Test` attend comme valeur la classe de l'exception qui devrait être levée.
- Exemple :

```
@Test(expected=IllegalArgumentException.class)
public void setNom() {
  personne.setNom("nom2");
  assertEquals("est ce que nom est correct", "nom2", personne.getNom());
  personne.setNom(null);
}
```

Le test de la levée d'exceptions (2/2)

- Si lors de l'exécution du test l'exception du type précisée n'est pas levée (aucune exception levée ou une autre exception est levée) alors le test échoue.
- Attention : l'utilisation de l'annotation ne permet que de vérifier que l'exception est levée.
- Pour vérifier des propriétés de l'exception, il est nécessaire d'utiliser le mécanisme utilisé avec JUnit 3 pour capturer l'exception et ainsi avoir accès aux membres de son instance.