# Junit 4 FrameWork

M. ETTIFOURI El Hassane ENSAO – 2017/2018

# Plan

- JUnit 4 vs JUnit 3
- Migration à partir de JUnit 3
- Rédaction d'une classe de test JUnit4
- Annotations : @Before et @After
- Particularités de @Test
- Tests paramétrables
- JUnit 4 dans Eclipse
- TP

# JUnit 3 vs JUnit 4

- Toutes les anciennes méthodes assertXXX sont les mêmes.
- Avec JUnit 4, il est plus facile de tester le déclanchement  des exceptions qui devraient être lancées par les méthodes testées.
- JUnit 4 peut  tester les tests du JUnit 3
- JUnit 4 offre une protection contre les boucles infinies
- JUnit 4 dispose  quelques fonctionnalités supplémentaires

# Migration à partir de JUnit 3

- JUnit 4 nécessite Java 5 ou supérieur
- Ne pas étendre junit.framework.TestCase, il suffit d'utiliser une classe ordinaire
- Importer org.junit. * et org.junit.Assert. * :
  - Utilisez de l'import *static* pour org.junit.Assert. *
  - Importations statiques remplacent l' héritage de junit.framework.TestCase
- Utilisez annotations lieu des noms de méthode spéciale:
  - Au lieu de la méthode setUp, mettez @Before avant une méthode quelquenques
  - Au lieu de la méthode tearDown, mettez @After avant une méthode quelquenques
  - Au lieu de commencer les noms de méthodes de test avec 'test', mettre @Test avant chaque méthode de test

# Rédaction d'une classe de test JUnit4 - I

- Importer les classes JUnit 4 dont vous avez besoin :
  - `import org.junit.*;`
    `import static org.junit.Assert.*;`
- Déclarez votre classe de la façon habituelle
  - `public class MyProgramTest {`
- Déclarer toutes les variables que vous allez utiliser fréquemment, y compris en général une instance de la classe à tester
  - `MyProgram program;`
    `int [] array;`
    `int solution;`

# Rédaction d'une classe de test JUnit4 - II

- Si vous le souhaitez, vous pouvez déclarer une méthode qui ne doit être exécutée qu'une seule fois, lors du chargement de la classe.
- C'est pour la configuration coûteuse, comme la connexion à une base de données
  - ```
    @BeforeClass
    public static void setUpClass() throws Exception {
    // méthode exécutée une seule fois, au début des tests
    }
    ```
- Si vous le souhaitez, vous pouvez déclarer une méthode qui ne doit être exécutée une seule fois, pour faire le nettoyage après la réalisation de tous les tests.
- ```
  @AfterClass
  public static void tearDownClass() throws Exception {
  // méthode exécutée une seule fois, à la fin des tests
  }
  ```

# Rédaction d'une classe de test JUnit4 - III

- Vous pouvez définir une ou plusieurs méthodes pour être exécutés avant chaque test; généralement ces méthodes initialisent les valeurs, de sorte que chaque test commence avec un nouveau jeu donnée.
  - ```
    @Before
    public void setUp() {
        program = new MyProgram();
        array = new int[] { 1, 2, 3, 4, 5 };
    }
    ```
- Vous pouvez définir une ou plusieurs méthodes pour être exécuté après chaque test; Typiquement, ces méthodes sont utilisés pour libérer les ressources, tels que les fichiers
  - ```
    @After
    public void tearDown() { program = null;
    }
    ```

# Annotations : @Before et @After

- Vous pouvez avoir autant de @Before et @After comme vous le souhaitez
  - Attention: Vous ne savez pas dans quel ordre elles vont exécuter
- Vous pouvez hériter les @Before et @After depuis une classe données, l'exécution sera donc :
  - Exécuter les méthodes @Before dans la superclass
  - Exécuter les méthodes @Before dans cette class
  - Executer une methode @Test de cette class
  - Exécuter les méthodes @After dans cette class
  - Exécuter les méthodes @After dans la superclass

# Rédaction d'une classe de test JUnit4 - IV

- Une méthode de test est annotée avec @Test, ne prend aucun paramètre et ne renvoie aucun résultat,
- Toutes les méthodes assertXXX habituels peuvent être utilisés
  - ```
    @Test
    public void sum() {
        assertEquals(15, program.sum(array));
        assertTrue(program.min(array) > 0);
    }
    ```
    ```

# Particularités de @Test

- Vous pouvez limiter la durée d'exécution d' une méthode
- Il s'agit d'une bonne protection contre les boucles infinie
- Le délai est spécifié en millisecondes
- Le test échoue si la méthode prend trop de temps
  - ```
    @Test (timeout=10)
     public void greatBig() {
        assertTrue(program.ackerman(5, 5) > 10e12);
    }
    ```
- Certains appels de méthode doit lever une exception
- Vous pouvez spécifier qu'une exception particulière est attendue
- Le test se passe si l'exception attendue est levée, et échoue autrement
  - ```
    @Test (expected=IllegalArgumentException.class)
    public void factorial() {
        program.factorial(-5);
    }
    ```

# Tests paramétrables

- À l'aide de @RunWith(value=Parameterized.class) et une méthode @Parameters, vous pouvez exécuter les mêmes tests avec plusieurs  jeux données

```
@RunWith(value=Parameterized.class)
public class FactorialTest {
 private long expected;
 private int value;

 @Parameters
 public static Collection data() {
    return Arrays.asList( new Object[ ][ ] { { 1, 0 }, { 1, 1 }, { 2, 2 }, { 120, 5 } });
 }

 public FactorialTest(long expected, int value) { // constructor
    this.expected = expected;
    this.value = value;
 }

 @Test
 public void factorial() {
    assertEquals(expected, new Calculator().factorial(value));
 }
}
```

# Ignorer un test

- L'annotation @Ignore indique qu' il ne faut pas exécuter un test
  - @Ignore(" sum(2,2) n'est pas prêt pour être testée pour le Sprint 1 !!")

```
@Test
public void add() {
    assertEquals(4, program.sum(2, 2));
}
```

- Le message associè à @Ignore doit expliquer la raison pour laquel on a ignoré le test, il sera affiché dans le rapport Junit.

# Test suites

- Comme précédemment, vous pouvez définir une série (suite) de tests

```
@RunWith(value=Suite.class)
@SuiteClasses(value={MyProgramTest.class,
                     AnotherTest.class})
public class AllTests { … }
```

# Autres informations

- Tests échoués lancent maintenant un AssertionError, plutôt que de AssertionFailedError de JUnit 3
- Il ya maintenant une version de assertEquals pour les tableaux d'objets:
  assertEquals(Object[] expected, Object[] actual)
  - Malheureusement, il n'existe toujours pas de assertEquals pour les tableaux primitives !!
- JUnit 3 avait une méthode assertEquals (p, p) pour chaque type p primitive, mais JUnit 4 a seulement une assertEquals (Object, Object) et dépend du caste de l' autoboxing

# Remarque importante

- La méthode suivante:
  - `long sum(long x, long y) { return x + y; }`
- avec le test suivant :
  - 
```
@Test
public void sum() {
    assertEquals(4, s.sum(2, 2));
}
```
- Donne un test KO :
  - `expected: <4> but was: <4>`
- Cela est dû à notre ami l' autoboxing:
  - `assertEquals` n'existe plus pour les primitives, existe uniquement pour les objets
  - Ainsi, le `4` est autoboxé à un `Integer`, alors que `sum` retourne un `long`
  - Le message d'erreur  signifie : int 4 est attendu, mais on a trouvé long 4
  - Pour que cela fonctionne, changer `4` par `4L`

# JUnit 4 dans Eclipse

- Comme d'habitude, la meilleure façon pour créer une classe de test est de laisser votre IDE le faire pour vous
- Voici l'approche recommandé pour les test-driven
  - D'abord , créez une classe contenant toutes les méthodes que vous pensez vous aurez besoin
  - Crèer  toutes les méthodes de test grace à votre IDE  (voir le TP  pour le cas d'eclipse)
  - répéter:
    - Écrire un test
    - Assurez-vous que le test échoue
    - Ecrire la méthode à tester
    - Assurez-vous que le test réussit maintenant