# JUnit

Description

JUnit est un framework pour écrire des tests. Il fait partie de la famille des frameworks xUnit. JUnit en est actuellement à sa version 4.

Exemple d'une classe de test

```
package com.vogella.junit.first;
import static org.junit.Assert.assertEquals;
import org.junit.AfterClass;
import org.junit.BeforeClass;
import org.junit.Test;
public class MyClassTest {
        @BeforeClass   public static void testSetup() {   }

        @AfterClass   public static void testCleanup() {
                // Teardown for data used by the unit tests
        }
        @Test(expected = IllegalArgumentException.class)
        public void testExceptionIsThrown() {
                MyClass tester = new MyClass();
                tester.multiply(1000, 5);
        }
        @Test   public void testMultiply() {
                MyClass tester = new MyClass();
                assertEquals("10 x 5 must be 50", 50, tester.multiply(10, 5));
        }
}
```

Installation

JUnit est maintenant intégré à Eclipse. Nous allons donc directement utiliser la version intégrée à Eclipse. La version que nous utiliserons est JUnit 4.

Création et exécution d'une classe de test avec JUnit

Pour créer une classe de test avec JUnit dans Eclipse
File → New → JUnit → JUnit Test case.

Pour executer une classe de test dans Eclipse
Run →Run As → JUnit Test

| **Annotation** | **Description** |
|---|---|
| @Test<br>public void method() | Cette annotation indique que la méthode est une méthode de test |
| @Before<br>public void method() | Cette annotation indique une méthode qui est exécutée avant chaque méthode de test. Cette méthode est une méthode spéciale qui permet de préparer l'environnement de test(initialisation, lecture des données…)<br>**Avant JUnit4, c'était la méthode spéciale setUp()** |
| @After<br>public void method() | Cette annotation indique une méthode qui est exécutée après chaque méthode de test. C'est une méthode spéciale qui permet de nettoyer l'environnement de test (suppression des données temporaires, restauration des données par défaut, …)<br>**Avant JUnit4, c'était la méthode spéciale tearDown()** |
| @BeforeClass<br>public static void method() | Cette annotation indique une méthode qui est exécutée une seule fois avant toutes les méthodes de test. Cette méthode peut donc être utilisée pour des initialisations communes à toutes les méthodes de test comme la connexion à une base de données. Une méthode annotée avec BeforeClass doit être déclarée static. |
| @AfterClass<br>public static void method() | Cette annotation indique une méthode qui est exécutée une seule fois après toutes les méthodes de test. Cette méthode peut donc être utilisée pour des activités de nettoyage telles que la déconnexion d'une base de données. Une méthode annotée avec BeforeClass doit être déclarée static. |
| @Ignore | Cette annotation permet d'ignorer la méthode test durant l'exécution des tests. Elle est utile lorsque le code à tester a changé mais que le test n'a pas encore été mis à jour. |
| @Test (expected = Exception.class) | La méthode test échoue si la méthode ne lance pas l'exception attendue. |
| @Test(timeout=100) | La méthode test échoue si l'exécution de la méthode dure plus que 100 millisecondes |

Template d'une méthode de test

```
@Test
public void testMultiply() {
 // MyClass is tested
 MyClass tester = new MyClass();
// Check if multiply(10,5) returns 50
assertEquals("10 x 5 must be 50", 50, tester.multiply(10, 5));
}
```

- JUnit 4 utilise plus les annotations par rapport à JUnit 3.
- Chaque méthode de test sera donc annotée avec @Test. Plus besoin de préfixer le nom de la méthode avec test comme c'était le cas dans JUnit3.
- Les méthodes de test dans JUnit ne sont pas exécutées forcement dans l'ordre ; elles ne doivent donc pas dépendre les unes des autres.
- Une méthode de test contient une ou plusieurs instructions fournies par le framework permettant de vérifier le résultat ou le comportement attendu pour la méthode ou l'objet testé à celui obtenu.

Annotations

Instruction de vérification (Asserts)

**Cf la javadoc de org.junit.Assert**

| **Instruction** | **Description** |
|---|---|
| fail(String) | Permet à la méthode de test d'échouer. Sert dans le test des exceptions |
| assertTrue([ String message], boolean condition) | Verifie que la condition booléenne est à vraie |
| assertsEquals([String message], expected, actual) | Vérifie que les valeurs *expected* et *actual* sont les mêmes. *actual* est la valeur retournée par le programme et *expected* celle attendue. |

| | |
|---|---|
| assertsEquals([String message], expected, actual, tolerance) | Idem que le précédent mais plus adapté aux types float et double. Le paramètre *tolerance* permet d'indiquer le nombre de décimales à prendre en compte dans la comparaison. |
| assertNull([String message], object) | Vérifie que l'objet en paramètre est null |
| assertNotNull([ String message], object) | Vérifie que l'objet en paramètre est non null |
| assertSame([String message], expected, actual) | Vérifie que les deux variables *expected* et *actual* sont des références du même objet |
| assertNotSame([ String message ], expected, actual) | Vérifie que les deux variables *expected* et *actual* sont des références d'objets différents |

NB. : Les messages dans les assertions doivent être expressifs afin d'aider à l'identification et à la résolution de problèmes.

**Quelques exemples**

Le test fail

```
public class Arithmetic {

	public static int divide(int a, int b){
		return a/b;
	}

}
```

```
@Test
	public void testInvalideInput() {
		try {
			int a=10;
			int b=0;
			Arithmetic.divide(a,);

			fail("Division by 0 should cause an exception");
		}
		catch (ArithmeticException exception) {
			exception.printStackTrace();
		}
```

```
	}
```

Le test échoue grâce à l'instruction **fail** si jamais l'exception attendue n'est pas levée et capturée dans le catch.
On peut écrire ce test en utilisant l'annotation @Test (expected=NomClasseException.class) mais dans ce cas on ne pourra pas préciser de message ni faire d'autres vérifications comme c'est le cas dans le bloc catch.

```
@Test(expected =ArithmeticException.class)
public void testInvalideInput() {
	int a=10;
	int b=0;
	Arithmetic.divide(a,);

}
```

*Différence entre assertSame et assertEquals*

```
import static org.junit.Assert.*;
import java.util.ArrayList;
import java.util.List;
import org.junit.Assert;
import org.junit.Test;

public class ExampleTest {

	@Test
	public void testSameObjects() {
		List l1=new ArrayList();
		l1.add("René");
		List l2=new ArrayList();
		l2.add("René");
		List l3=l2;
		Assert.assertEquals("l1 et l2 sont des objets différents mais
qui ont la même valeur",l1, l2);
		Assert.assertNotSame("l1 et l2 sont des objets différents; ils
n'ont donc pas la même reférence; modifier l1 n'impactera pas l2 et vice
et versa",l1, l2);
		Assert.assertSame("l2 et l3 representent le même objet (ils ont
la même reférence); Modifier l2 c'ets modifier l3 ",l3, l2);


	}

}
```

Si on modifie la valeur de l2 et qu'on refait le test, qu'est ce qui se passera?
NB.: La comparaison des listes est possible car la methode equals est implémentée dans la classe ArrayList et aussi dans la classe String.

Suite de tests

Lorsqu'on a plusieurs classes de test, on peut le regrouper dans une suite de tests. L'exécution de la suite de test exécutera toutes les classes de test qui la composent.

Exemple de suite de tests composée de deux classes de test. Le code est placé dans la classe AllTests

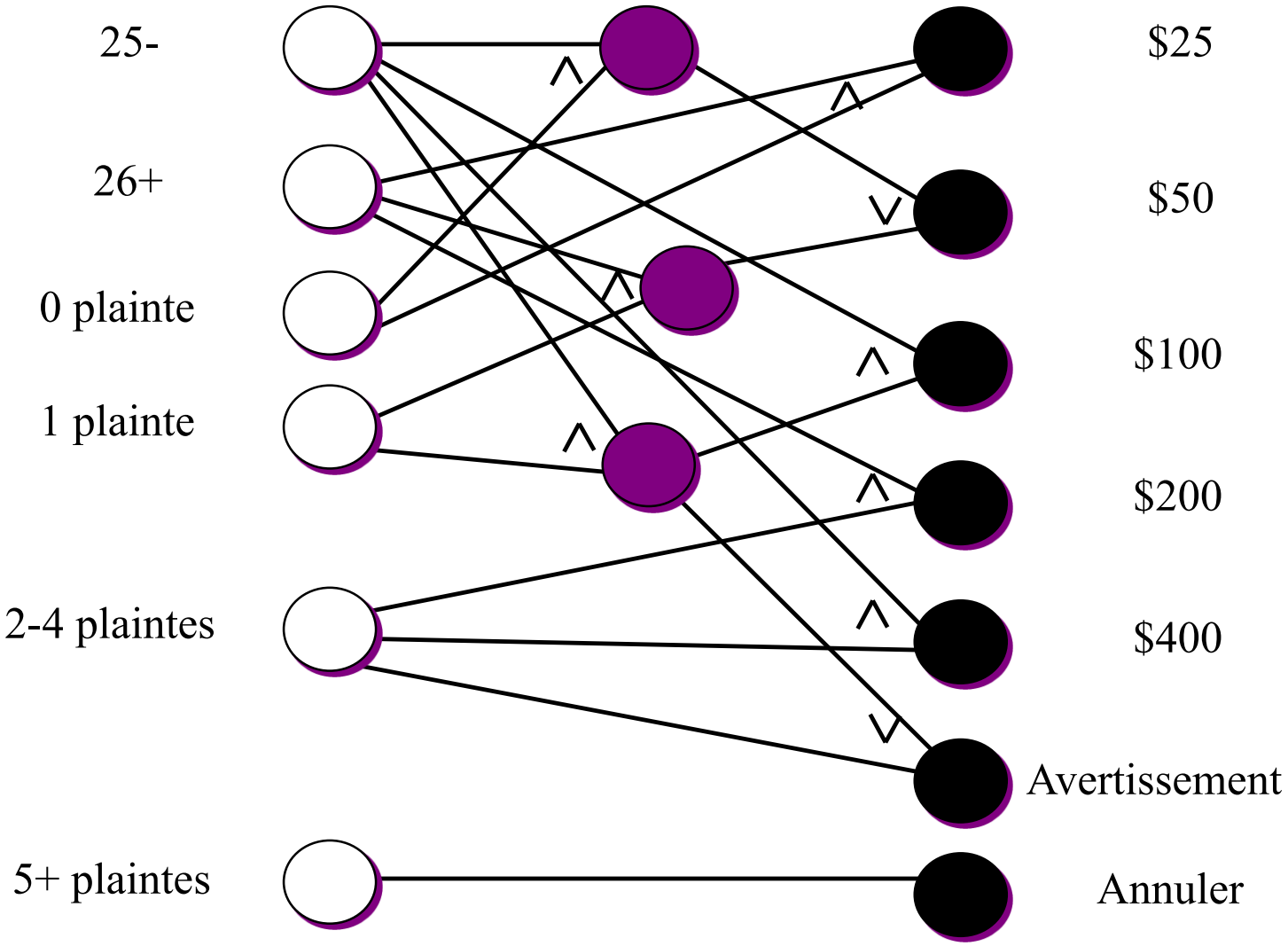
```
package com.vogella.junit.first;
import org.junit.runner.RunWith;
import org.junit.runners.Suite;
import org.junit.runners.Suite.SuiteClasses;
@RunWith(Suite.class)
@SuiteClasses({ MyClassTest.class, MySecondClassTest.class })
public class AllTests { }
```

**Exercices**

**Exercice 1 (Lire les spécifications des méthodes et tester tous les cas de figure)**
Ecrire un test unitaire **StringBufferTest** permettant de tester les méthodes charAt, setCharAt et append(String str) de la classe StringBuffer (http://java.sun.com/j2se/1.4.2/docs/api/java/lang/StringBuffer.html) du Java SDK. (Définir également les méthodes de test testAppendWithNullString et testCharAtWithInvalidIndexes testant les exceptions susceptibles d'être levées par la méthode charAt.)

**Exercice 2**

Sur la base du graphe de cause à effet suivant, écrivez et testez par classe d'équivalence forte le programme Java qui gèrerait le renouvellement d'assurance pour des clients selon leur âge et le nombre de plaintes.




**Références**
**http://deptinfo.cnam.fr**
http://www.vogella.com/articles/JUnit/article.html
http://www.eclipsetotale.com/articles/premierPas.html (Premiers pas avec Eclipse pour voir comment créer un projet et se familiariser avec l'environnement)