#### FRAMEWORK JUNIT



#### PLAN DU COURS

- Introduction
- Tests Unitaires
- Utilisation de JUNIT
  - TestCase
  - Annotations
- Place à la Pratique

#### INTRODUCTION

- Il existe différents niveaux de test :
  - Test unitaire
  - Test d'intégration
  - Test de charge
  - Test fonctionnel
  - Test sécurité

**—** ....

# TEST UNITAIRE: DÉFINITION

 Un test unitaire est une procédure permettant de vérifier le bon fonctionnement d'une partie précise d'un logiciel. Il s'agit d'un code.

En POO, on teste au niveau des classes

 Pour chaque classe (MyClass), on a une classe de test (MyClassTest).

# TEST UNITAIRE: QUELQUES RÈGLES

• Doit être isolé : il doit être indépendant

 N'est pas un test de bout en bout : il agit que sur une portion de code

 Doit être déterministe : le résultat doit être le même pour les mêmes entrées

Est le plus petit et simple possible

# TEST UNITAIRE: QUELQUES RÈGLES

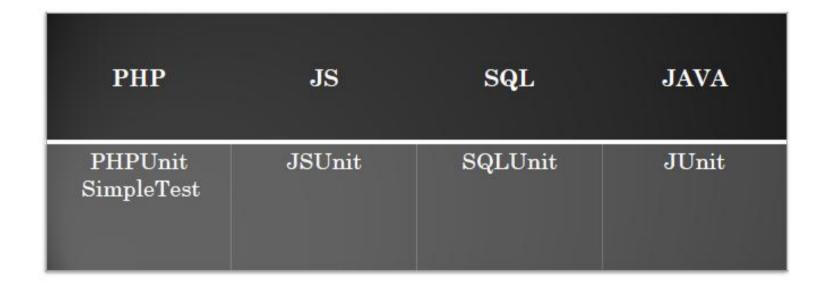
- Ne teste pas d'enchainement d'actions
- Etre lancé le plus souvent possible : intégration continue
- Etre lancé le plus tôt possible : détection des bug plus rapide
- Couvrir le plus de code possible
- Etre lancé a chaque modification

## TEST UNITAIRE: AVANTAGE ET INTÉRÊT

- Garantie la non régression
- Détection de bug plus facile
- Aide a isoler les fonctions
- Aide a voir l'avancement d'un projet (TDD)

<sup>\*</sup> Le test-driven development (TDD) ou en français développement piloté par les tests est une technique de développement de logiciel qui préconise d'écrire les tests unitaires avant d'écrire le code source d'un logiciel.

## TEST UNITAIRE: OUTIL DE TEST



#### TEST UNITAIRE: CAS A TESTER

 Lors de l'utilisation de test unitaire on se doit de tester différents cas.

- Cas en succès : fonctionnement normal
- Cas d'erreur : test sur la gestion d'erreur

)

# TEST UNITAIRE : LES RÉSULTATS

• Un test unitaire peux renvoyer 3 résultats différents :

- Success: test réussi

Error : erreur inattendue a l'exécution

- Failure: au moins une assertion est fausse

10

#### TEST UNITAIRE: MOCK

- Quelques fois un test a besoin d'un composant donné pour s'exécuter.
- Par exemple pour tester une fonctionnalité, nous avons besoin du retour d'un Web Service, qui n'a toujours pas été développé.
- Il est alors utile d'utiliser des bouchons (MOCK) pour isoler le test.
- De plus un bouchon permet de tester tout les cas (valeur correcte, erroné etc.)

1 1

 Il n'y a pas de limite au nombre de tests au sein de notre classe de test.

 On écrit au moins un test par méthode de la classe testée.

• Pour désigner une méthode comme un test, il suffit d'utiliser l'annotation **@Test** (a partir de JUnit4).

• Au sein des tests, on utilise des assertions pour valider ou non un test. Quelques assertions indispensable:

Assertion	Action
assertEquals()	Vérifie l'egalité entre deux entités
assertNotEquals()	Vérifie l'inégalité entre deux entités
assertFalse()	Vérifie que la valeur fourni en paramètre est fausse
assertTrue()	Vérifie que la valeur fourni en paramètre est vrai
assertNull()	Vérifie que la valeur fourni en paramètre est l'objet NULL
assertNotNull()	Vérifie que la valeur fourni en paramètre n'est pas l'objet NULL

© 2017 **JUNIT** 

```
package com.m2i.formation;
import junit.framework.TestCase;
public class AdditionTest2 extends TestCase {
protected Addition op = new Addition();
   public void testCalculer() throws Exception {
       assertEquals(5, op.calculer(1, 3));
   public void testLireSymbole() throws Exception {
       assertEquals('-', op.lireSymbole());
```

```
package com.m2i.formation;
import static org.junit.Assert.assertEquals;
import org.junit.Test;
public class AdditionTest {
   protected Addition op = new Addition();
   @Test
   public void testCalculer() throws Exception {
           assertEquals(5, op.calculer(1, 3));
   @Test
   public void testLireSymbole() throws Exception {
           assertEquals('-', op.lireSymbole());
```

15

## JUNIT: PLACE A LA PRATIQUE

Développement d'un Projet qui implémente JUNIT Ajout de JUNIT dans un Projet déjà existant