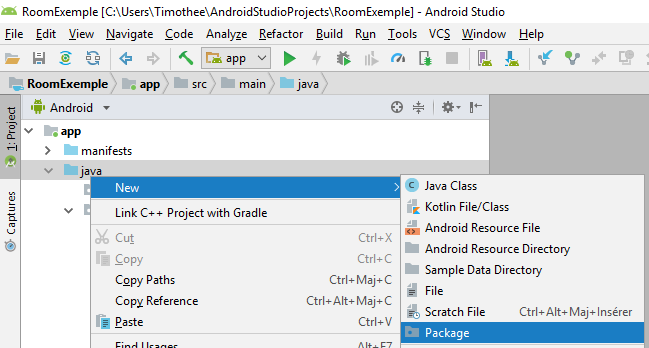
**Contexte 1ère partie** : vous êtes en stage dans une boulangerie où vous êtes chargés de développer une calculatrice enregistreuse pour remplacer la caisse enregistreuse dont la maintenance est onéreuse. La boulangère, méfiante, vous demande de tester soigneusement votre application avant d'essayer le pilote

**Partie 1**

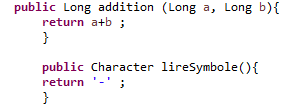
**Exercice n°1 : test d’un programme calculatrice**

Etape 1: créer un nouveau package Java, dans le répertoire app\java. Appelez le par exemple **calculatrice**

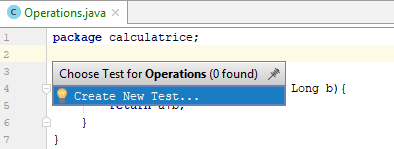


Etape 2: créez une classe nommée Operations, en se plaçant dans le package **calculatrice** que vous venez de créer.

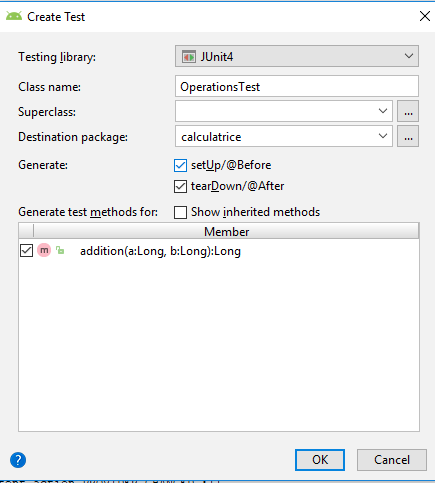
Etape 3: ajoutez les 2 méthodes suivantes à la classe Operations :



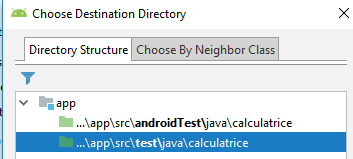
Etape 4: une fois le code entré, mettez-vous n’importe où dans le code et tapez : Ctrl+Shift+T



Etape 5 : ajoutez un setup et teardown



Etape 6 : conservez le choix par défaut (test unitaire)

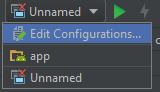


Remarque : le répertoire **test** stocke les tests unitaires tandis que le répertoire **androitTest** stocke les tests unitaires ***instrumentés*** c’est-à-dire sur le device.

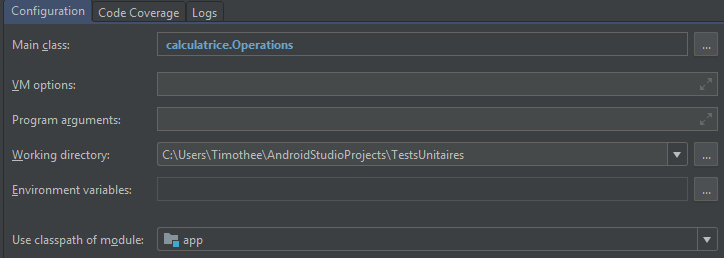
Etape 6: Faire un clic droit dans le fichier OperationsTest.java et sélectionnez **Run OperationsTest with coverage**

Pour pouvoir exécuter ce programme dans Android Studio (IntelliJ) il faut préalablement définir une configuration de type application :

* Cliquez sur Edit configurations



* Remplissez les éléments correspondant à Main class et Use classpath of module :



**Question 1**: quel est le résultat du test ? Pourquoi ?

Etape 7: on ajoute le code suivant à la classe OperationsTest.java

Une variable d'instance:

**public** Operations op;

**Question 2**: où faut il ajouter cette déclaration ? qu'est ce que op ?

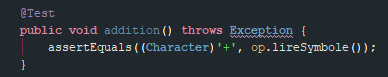
Etape 10: on ajoute le code suivant à la classe OperationsTest.java

--> dans la méthode setUp()

op = **new** Operations() ;

**Question 3**: à quoi sert cette instruction ?

Etape 8: on ajoute un test , par exemple pour tester l’opération d’addition : ajoutez le code suivant



Etape 9: Faire un clic droit dans le fichier OperationsTest.java et sélectionnez de nouveau Run OperationsTest with coverage

**Question 4**:

* quel est le résultat du test ? Pourquoi ?
* comment corriger le problème ? Effectuez la correction.

Etape 10: dans le fichier OperationsTest.java, on ajoute un nouveau test

@Test

**public** **void** testCalculer() **throws** Exception {

*assertEquals*(**new** Long(4),

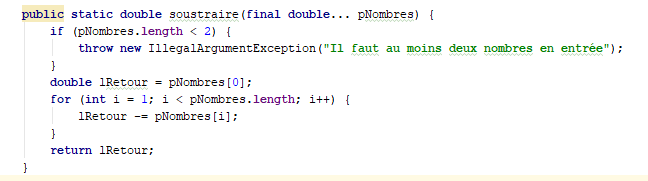
op.addition(**new** Long(1), **new** Long(3)));

}

**Question 5**:

* Où placer ce test ? Exécutez les tests et commenter les résultats. Que font ces tests ?
* Comment Junit sait-il qu'il faut exécuter ces méthodes (testCalculer() et addition() ) ?

Etape 11:nous allons ajouter dans la classe Operations une méthode permettant d'effectuer des soustractions à partir de 2 ou plusieurs nombres



**Point de cours**:

- le terme **static** pour la méthode soustraire signifie que c'est une méthode de classe

- le terme **final** dans le paramètre de la méthode *soustraire* signifie que la valeur du paramètre qui suit ne peut pas être modifié par la suite

- le terme ... dans le paramètre de la méthode *soustraire* signifie que le paramètre pNombres est un tableau de 1 à N éléments

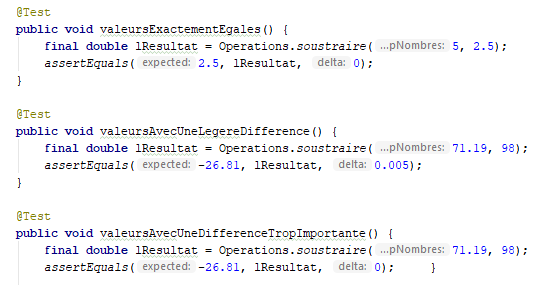
**Question 6**:

- que fait cette méthode ? que retourne t'elle ?

**Rappel**: comme la méthode soustraire est une méthode de classe (static), elle va s'appliquer à la classe et non un objet, de la forme *Operations.soustraire* (à compléter).

Etape 12:nous allons créer dans des tests unitaires permettant de mesurer la précision de cette calculatrice pour s'assurer qu'elle ne fait pas perdre des centimes à la boulangère.

Ajoutez 3 tests à la classe OperationsTest



**Question 7**:

* quel est le résultat des tests ? que font ces tests ?
* ajoutez un test qui soustrait 4 valeurs: 5 euros, la valeur d'un billet, 0.65, 1.10, 0.15 les valeurs d'une baguette, d'une baguette tradition et d'un bonbon.
* A t'on une précision exacte ? Que faut-il faire ? Qu'en concluez vous ?

**Partie 2**

**Contexte 2ème partie** : vous êtes en stage dans une SSII qui vous demande de mettre au point un ensemble de tests permettant de vérifier la qualité et l'homogénéité du code des développeurs Junior en Java, qui programmaient auparavant en PHP. Cette vérification va passer par le contrôle ***du respect des contrats*** et l'utilisation de ***suppositions***. Nous allons utiliser JUnit et également une librairie externe, hamcrest

**Exercice n°2 : test de l'implémentation Java**

Etape 0 cette fois ci, nous allons créer directement des programmes de tests, qui vont vous permettre de former l'équipe aux nouveaux principes de programmation.

-> créez un nouveau test unitaire JUnit, File --> New --> JUnit Test Case : CTRL + SHIFT + T , donez lui un autre nom.

Etape 1 :

Dans le fichier **build.gradle** (module :app), ajoutez la ligne suivante dans la partie **dependancies** :

testImplementation 'org.hamcrest:hamcrest-library:1.3'

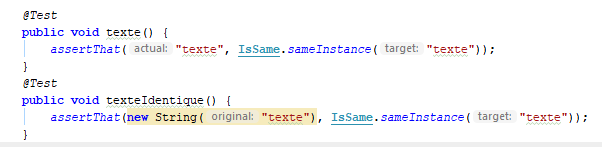
* Cela va permettre d’ajouter la librairie d’assertions hamcrest au projet via le gestionnaire de dépendances Gradle.

A propos d’hamcrest : <https://www.vogella.com/tutorials/Hamcrest/article.html>

Etape 2: une fois le programme généré, ajoutez dans la partie des imports la ligne suivante, qui permet d'importer la librairie externe.

**import** org.hamcrest.core.\*;

Etape 3: créez les 2 tests suivants



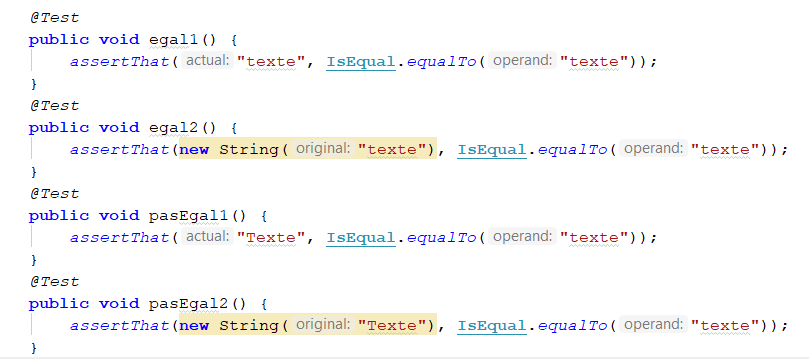
A ce stade, les 2 méthodes assertThat seront en rouge et vous pouvez corriger en faisant un import static.

**Question 1**: analyse des Contrats permettant de savoir si 2 objets sont identiques

1.a: Que font ces tests ?

1.b: Pouvez vous expliquer les résultats ?

Etape 4: ajoutez les 4 tests suivants



**Question 2**: analyse des Contrats permettant de savoir si le contenu de 2 objets est identique

2.1: Que font ces tests ?

2.2: Pouvez vous expliquer les résultats ?

**Point de cours**:

Une supposition vérifie une condition. Si la condition n'est pas vérifiée, le test s'arrête mais ne passe pas en erreur ou en échec.

Les différentes suppositions sont :

• assumeNoException(java.lang.Throwable) : vérifie qu'une opération s'est déroulée sans lever de Throwable.

• assumeNotNull(java.lang.Object...) : vérifie qu'aucun des paramètres n'est à null.

• assumeThat(T, org.hamcrest.Matcher) : vérifie qu'une condition définie par contrat est respectée (voir "Assertion sur une condition définie par contrat").

• assumeTrue(boolean) : vérifie que le paramètre est vrai.

**Partie 3 :** tests de collections

Cf rappels méthodologiques pour les collections en Java.

**Exercice n°3 : test de collections**

Etape 1: dans Android Studio, créez un package navigation et dedans une classe Marin comme dans le doc méthodo.

Ajoutez un getter et setter.

Etape 2: générez une classe de test unitaire testMarin

Ctrl+Shift+T dans la classe marin, nommez par exemple la classe testMarin

Etape 3 : génération d’une collection

Ici on ne va pas vraiment tester la classe Marin mais une collection, et on va s’en servir par la suite.

****

**Question 3**: commentez. Combien de tests effectués ? Comment se termine le test ?

**Question 4**: créez une classe Navire avec

* Un attribut nommé equipage qui est une liste de Marin
* Ajoutez 2 constructeurs :
  + Un vide
  + Un avec en paramètre un Equipage
* Ajoutez le getter et setter
* Ajoutez une méthode permettant **d’ajouter** un **Marin** à l’équipage
* Créez une classe de test de la classe Navire
* Testez la méthode d’ajout de marin
  + Vérifiez que l’équipage au début est vide au début (méthode emptyIterable)
  + Ajoutez un Marin à, l’équipage et vérifiez alors que l’équipage contient bien ce marin.
* Demandez le supplément : exercice sur différence entre contains et hasItems