

Compte rendu – Tests Android – SLAM5.9

Introduction	1
Conception de la solution	
Application jouet	2
Tests unitaires – <i>Convertisseur.java</i>	4
Tests unitaires – <i>MainActivity.java</i>	11
Tests de l'interface	14
Ressenti personnel	16

Introduction

Objectif : L'objectif de ce TD est de comprendre et de réaliser des tests unitaires. Ces derniers nous permettent de coder au mieux une application et d'éviter tout bug qui ferait planter l'app. Ainsi, nous programmons des tests unitaires, d'intégration et d'instrumentation sur Android.

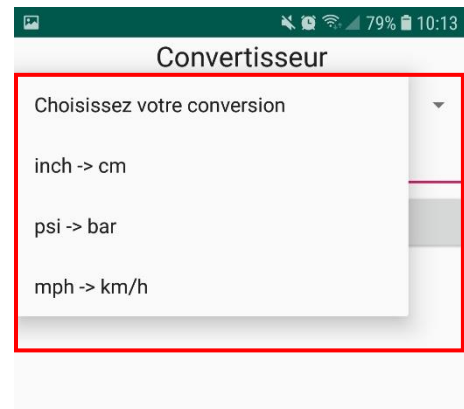
Mise en situation : Nous travaillons sur une application très simple, un convertisseur d'unités : dimensions, monnaie, etc.... L'utilisateur saisit une valeur, puis un mode de conversion et clique sur un bouton pour avoir la valeur convertie.

Résolution : Nous travaillons en java sur la plateforme *AndroidStudio* et utilisons notre téléphone portable comme émulateur AVD.

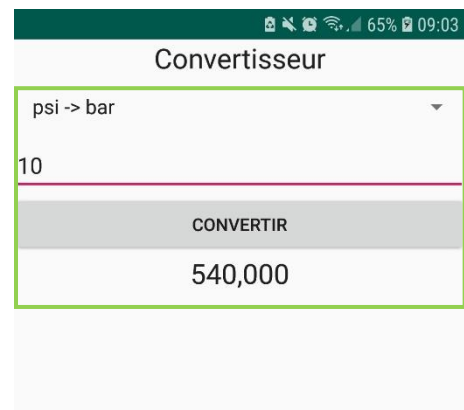
Conception de la solution

- Application jouet

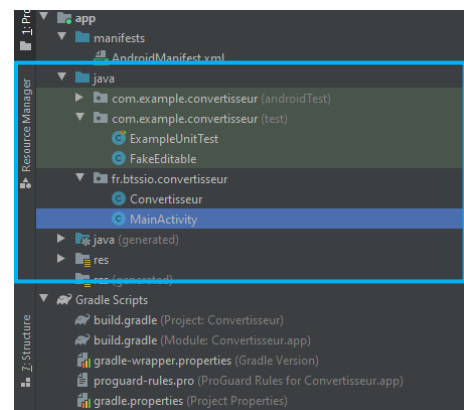
L'application est toute simple, elle est composée d'une zone de saisie et d'un menu déroulant qui permet de choisir le type de conversion à effectuer (**rouge**).



Au stade actuel, on remarque que l'application ne fonctionne pas correctement puisque lors de la conversion de 10 psis on obtient 540 bars (contre 0.689 en temps normal) (**vert**).



Pour obtenir ce résultat on organise le menu de notre application de la manière ci-contre (**bleu**) en veillant à surtout bien conserver les dossiers de test.



Reste à ajouter les dépendances qui seront nécessaires au déroulement du td dans le fichier build.gradle de l'app (rose).

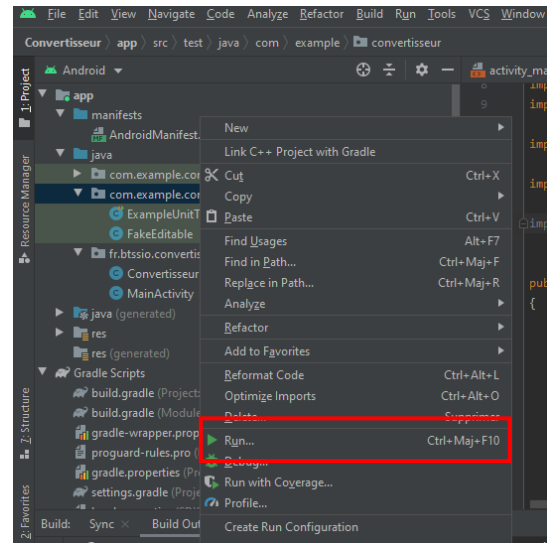
```
dependencies {

    implementation 'androidx.appcompat:appcompat:1.2.0'
    implementation 'com.google.android.material:material:1.2.1'
    implementation 'androidx.constraintlayout:constraintlayout:2.0.4'
    testImplementation 'junit:junit:4.+'
    androidTestImplementation 'androidx.test.ext:junit:1.1.2'
    androidTestImplementation 'androidx.test.espresso:espresso-core:3.3.0'

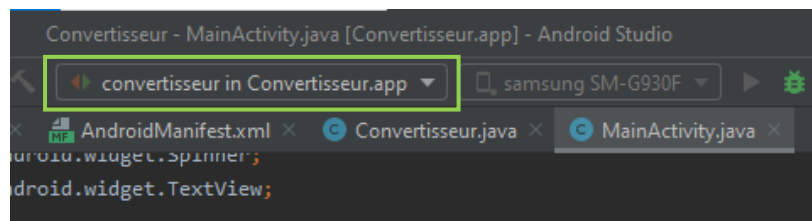
    testImplementation 'junit:junit:4.12'
    testImplementation 'org.hamcrest:hamcrest-library:1.3'
    testImplementation 'pl.pratismatists:JUnitParams:1.1.1'
    androidTestImplementation 'androidx.test.ext:junit:1.1.1'
    androidTestImplementation 'androidx.test.runner:1.2.0'
    // mockito : objets bidons pour faciliter les tests
    testImplementation 'org.mockito:mockito-core:2.25.0'
    androidTestImplementation 'org.mockito:mockito-android:2.25.0'
    // fuites mémoire : références d'objets Android gardées à tort
    debugImplementation 'com.squareup.leakcanary:leakcanary-android:2.3'
    // espresso : tests de l'interface avec un AVD
    androidTestImplementation 'androidx.test.espresso:espresso-core:3.2.0'
    androidTestImplementation 'androidx.test.espresso:espresso-intents:3.2.0'
    androidTestImplementation 'androidx.test.espresso:espresso-web:3.2.0'
}
```

- Tests unitaires – *Convertisseur.java*

Cette partie a pour but de montrer de quelle façon les tests sur une application sont mis en place. Pour lancer un test, il faut tout d’abord cliquer droit sur la classe en question puis choisir *Run* (rouge).

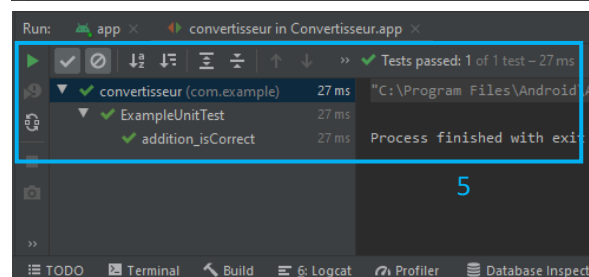
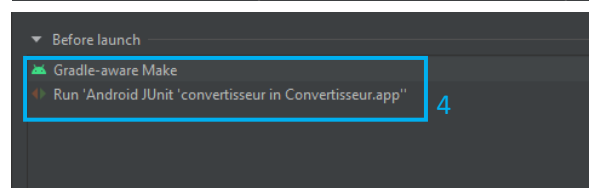
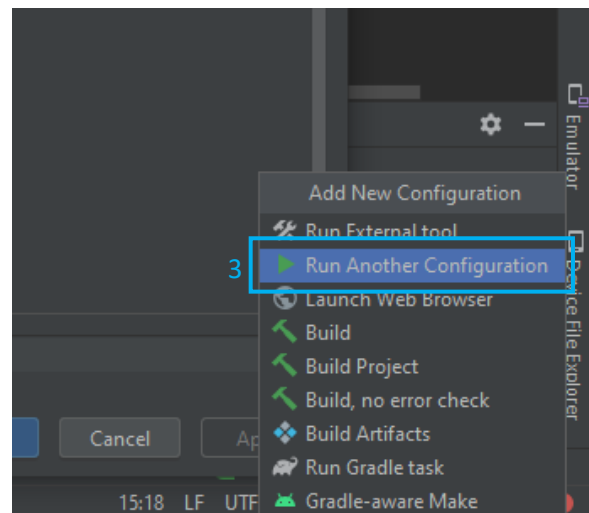
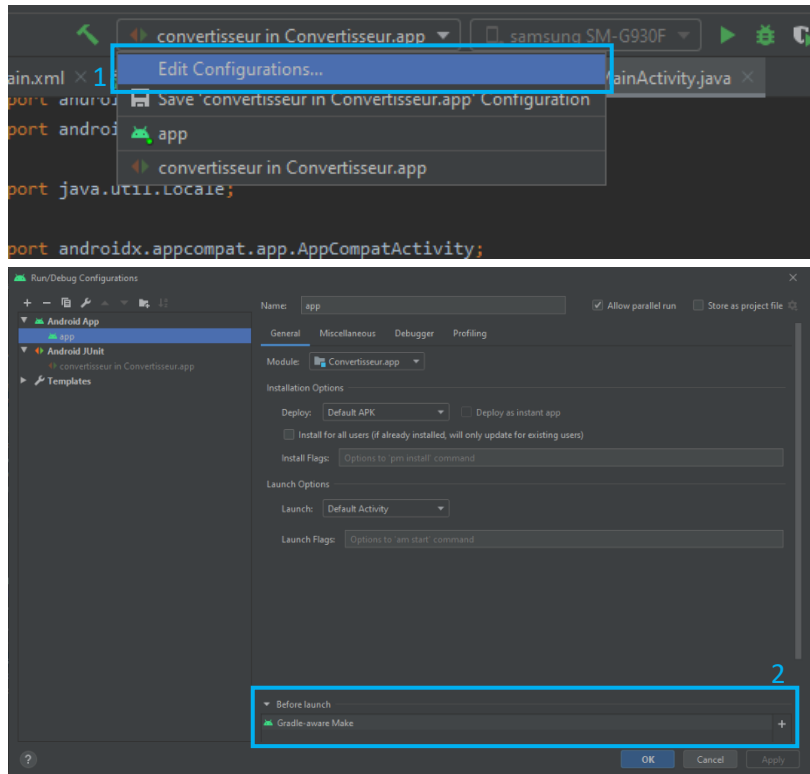


Un nouvel item apparait afin de lancer le test (vert).



Il est ensuite nécessaire de configurer le lancement de l'application de telle sorte que les tests soient effectués en premier lieu. On suit les 4 étapes numérotées ci-contre (bleu).

1. Dérouler le menu de l'item et sélectionner *Edit Configurations*
2. Dans la zone *Before launch* cliquer sur le plus
3. Puis choisir *Run Another Configuration*
4. Sélectionner le test et le visualiser dans l'espace de pré-lancement
5. Si on lance le test tout est au vert.



On test ensuite le constructeur. Pour cela, on crée la méthode suivante `constructeurDoitInitCodeAZero()`. La construction de ce test est essentielle à comprendre puisque tous les autres tests seront réalisés de la même façon (rose).

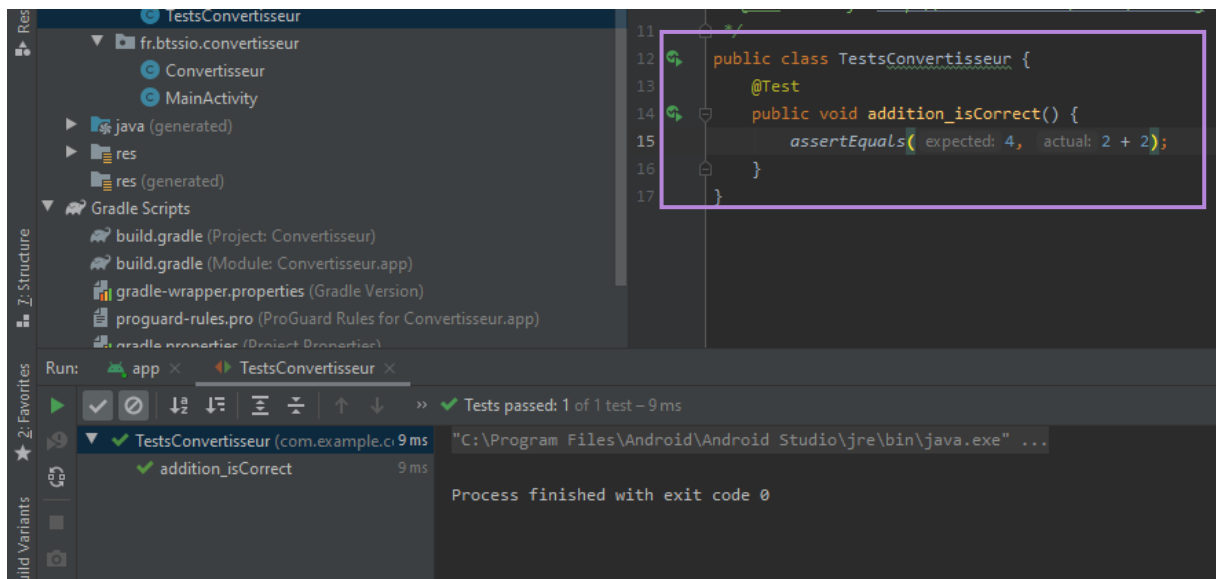
arrange : permet de préparer les données qui seront traitées.

act : correspond à l'action qui sera effectuée par le test (souvent l'appel d'une méthode).

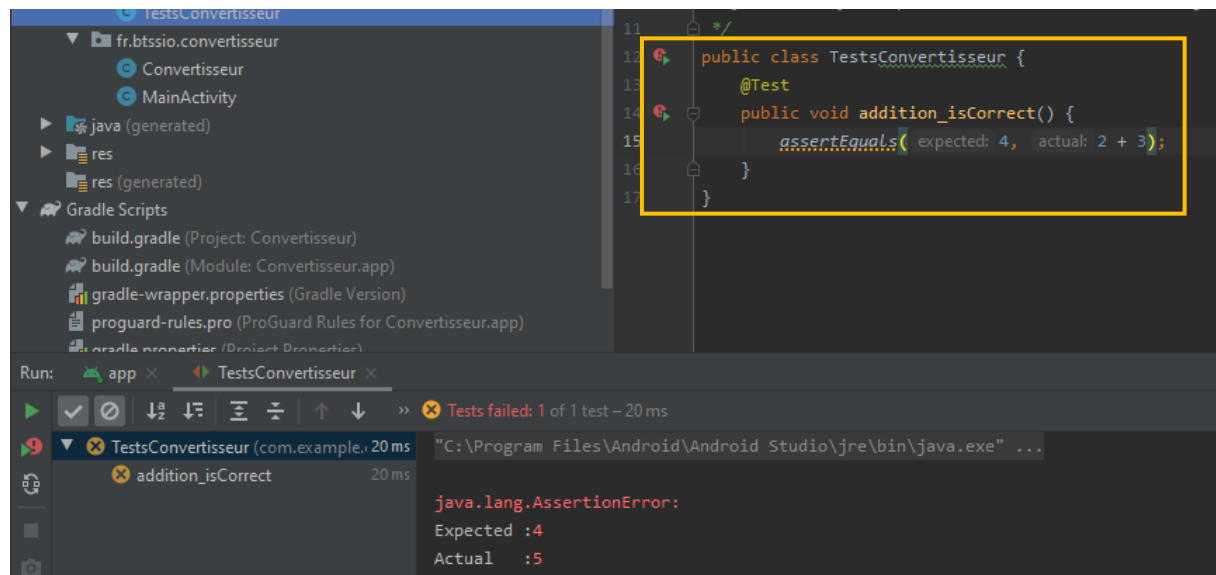
assert : vérifie si le résultat trouvé est celui attendu. Il existe plusieurs façons de vérifier (*JUnit* ou *Hamcrest*).

```
32      @Test
33      public void constructeurDoitInitCodeAZero()
34      {
35          /// arrange : préparer les données
36          Convertisseur convertisseur = new Convertisseur();
37          /// act : faire une action cruciale pour ce test
38          int code = convertisseur.getCodeConversion();
39          /// assert : vérifier que le résultat est conforme
40          // variante avec assert (JUnit "pur")
41          assertEquals( message: "le code initial doit être 0", expected: 0, code);
42      }
```

On test que tout fonctionne grâce à la méthode suivante (violet) : $4 \text{ (expected)} = 2 + 2 \text{ (actual)}$?

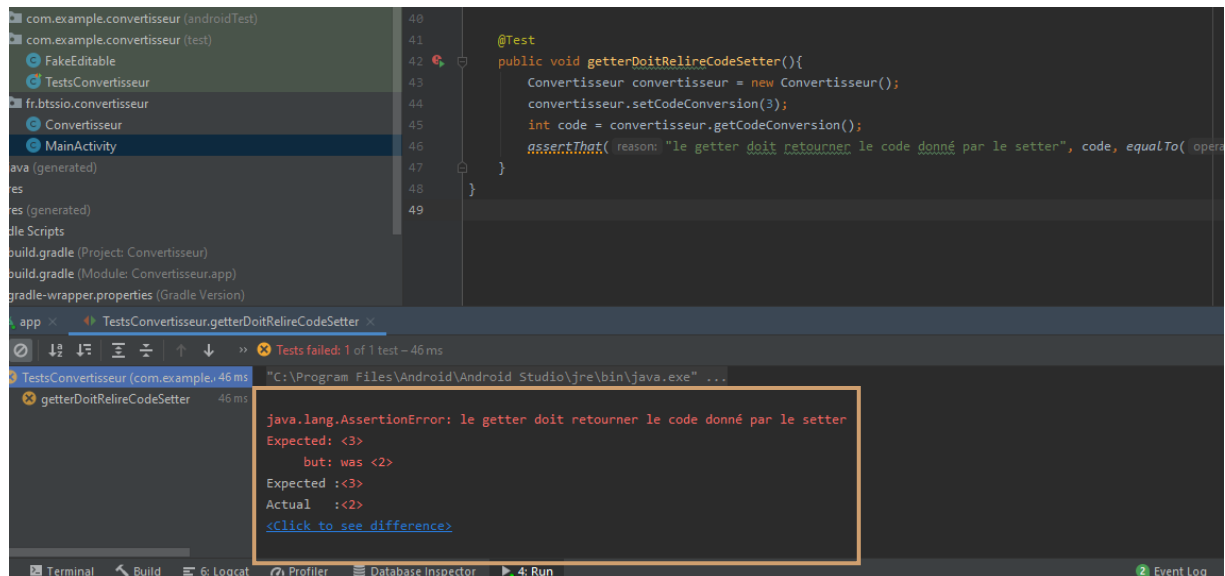


Au contraire, en cas d'erreur (orange) : $4 = 2 + 3$? Le logiciel compare dans la console le résultat trouvé par la machine et celui attendu.

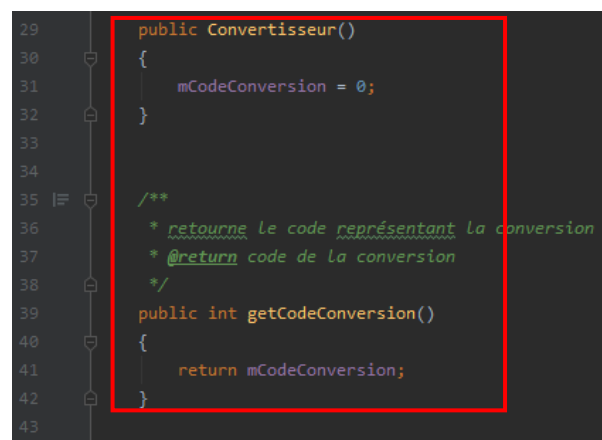


Maintenant on souhaite vérifier que le setter et le getter fonctionnent bien. En suivant l'exemple précédent, nous écrivons un test `getterDoitRelireCodeSetter` qui vérifie le couple setter-getter.

Pour cela, on crée un convertisseur, on utilise le setter pour mettre le code à 3 puis on récupère la valeur donnée par le getter dans une variable. Enfin on vérifie que la valeur du getter vaut 3. Comme prévu on se retrouve avec une erreur (marron) car des bugs sont signalés dans le td.



Pour corriger ceci on modifie le constructeur et le getter comme ci-contre (rouge).



Afin de vérifier que ce n'est pas seulement un coup de chance, on choisit de créer un test qui valide plusieurs valeurs successivement comme ci-après. On crée pour cela une collection qui est passée en revue où chaque valeur est vérifiée (vert).

```

54  @Test
55  @Parameters
56  public void getterDoitRelireCodeSetterMultiple(int codeSet)
57  {
58      /// arrange : préparer les données
59      Convertisseur convertisseur = new Convertisseur();
60      /// act : faire une action cruciale pour ce test
61      convertisseur.setCodeConversion(codeSet);
62      int codeGet = convertisseur.getCodeConversion();
63      /// assert : vérifier que le résultat est conforme
64      assertThat("le getter doit retourner le code donné par le setter par multiple", codeGet, equalTo(codeSet));
65  }
66
67  static Collection<Object[]>
68  parametersForGetterDoitRelireCodeSetterMultiple() {
69      return Arrays.asList(new Object[][]{
70          {0}, {1}, {2}, {3}, {4}, {5}
71      });
72  }

```

On corrige le setter en conséquence (bleu) mais on remarque que les valeurs qui dépassent le nombre d'item dans le menu ne passent pas (rose).

```

44  /**
45   * définit la prochaine conversion à faire
46   * @param codeConversion code de la conversion
47   */
48  public void setCodeConversion(int codeConversion) throws IllegalArgumentException
49  {
50      if (codeConversion < 0 || codeConversion > mCoefficients.length) { // BUG à
51          throw new IllegalArgumentException();
52      }
53      this.mCodeConversion = (int) (codeConversion*2- codeConversion); // TODO BUG
54  }
55
56

```

```

getterDoitRelireCodeSetterMultiple 35 ms
✓ TestsConvertisseur.getterDoitRelireCodeSetterMul 19 ms
✓ TestsConvertisseur.getterDoitRelireCodeSetterMul 0 ms
✓ TestsConvertisseur.getterDoitRelireCodeSetterMul 0 ms
✓ TestsConvertisseur.getterDoitRelireCodeSetterMul 0 ms
! TestsConvertisseur.getterDoitRelireCodeSetterMul 15 ms
! TestsConvertisseur.getterDoitRelireCodeSetterMul 1 ms
getterDoitConvertirInchVersCm

```

On passe désormais aux tests de conversions d'unités. Ils suivent tous le même modèle que précédemment. Voyez ci-dessous la validation de la conversion d'inch en cm (**violet**), de psi en bar (**orange**) et de mph en kmh (**marron**).

```

74  @Test
75  @Parameters
76  ▶ public void getterDoiVerifierConversionInchVersCm(double codeSet)
77  {
78      /// arrange : préparer les données
79      Convertisseur convertisseur = new Convertisseur();
80      /// act : faire une action cruciale pour ce test
81      convertisseur.setCodeConversion(0);
82      double code = convertisseur.convertir(codeSet);
83      /// assert : vérifier que le résultat est conforme
84      assertThat(reason: "inch to cm succeed for multiple values", code, equalTo(operand: codeSet*2.54));
85  }
86  static Collection<Object[]>
87  @ parametersForGetterDoiVerifierConversionInchVersCm() {
88      return Arrays.asList(new Object[][]{
89          {0.00}, {1.00}, {13.00}, {-1.18}
90      });
91  }

```

```

05  @Test
06  @Parameters
07  ▶ public void getterDoiVerifierConversionPsiVersBar(double codeSet)
08  {
09      /// arrange : préparer les données
10      Convertisseur convertisseur = new Convertisseur();
11      /// act : faire une action cruciale pour ce test
12      convertisseur.setCodeConversion(1);
13      double code = convertisseur.convertir(codeSet);
14      /// assert : vérifier que le résultat est conforme
15      assertThat(reason: "inch to cm succeed for multiple values", code, equalTo(operand: codeSet*0.0689476));
16  }
17  static Collection<Object[]>
18  @ parametersForGetterDoiVerifierConversionPsiVersBar() {
19      return Arrays.asList(new Object[][]{
20          {0.00}, {1.00}, {44.00}, {-7.25}
21      });
22  }

```

```

34  @Test
35  @Parameters
36  ▶ public void getterDoiVerifierConversionMphVersKmh(double codeSet)
37  {
38      /// arrange : préparer les données
39      Convertisseur convertisseur = new Convertisseur();
40      /// act : faire une action cruciale pour ce test
41      convertisseur.setCodeConversion(2);
42      double code = convertisseur.convertir(codeSet);
43      /// assert : vérifier que le résultat est conforme
44      assertThat(reason: "mph to kmh succeed for multiple values", code, equalTo(operand: codeSet*1.609344));
45  }
46  static Collection<Object[]>
47  @ parametersForGetterDoiVerifierConversionMphVersKmh() {
48      return Arrays.asList(new Object[][]{
49          {0.00}, {1.00}, {160.00}, {-37.28}
50      });
51  }

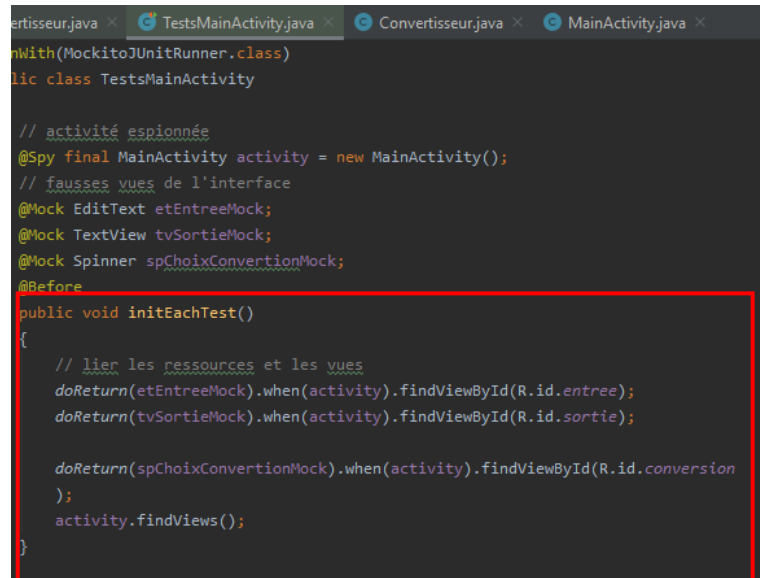
```

- Tests unitaires – *MainActivity.java*

Cette partie a pour but de montrer de quelle façon les tests pour simuler des vues sont effectués. Etant donné que nous n'avons pas l'environnement Android complet, nous simulons certaines choses à l'aide de Mockito.

On commence par créer la classe ci-contre qui va permettre d'analyser toutes les vues (**rouge**). L'idée est que l'activité aille chercher le texte saisi dans *etEntree*.

Ensuite, l'activité effectue la conversion puis affiche le résultat dans une variable dont on peut vérifier le contenu.



```
import org.junit.runner.RunWith;
import org.mockito.Mock;
import org.mockito.junit.MockitoJUnitRunner;

@RunWith(MockitoJUnitRunner.class)
public class TestsMainActivity {

    // activité espionnée
    @Spy final MainActivity activity = new MainActivity();
    // fausses vues de l'interface
    @Mock EditText etEntreeMock;
    @Mock TextView tvSortieMock;
    @Mock Spinner spChoixConversionMock;

    @Before
    public void initEachTest()
    {
        // lier les ressources et les vues
        doReturn(etEntreeMock).when(activity).findViewById(R.id.entree);
        doReturn(tvSortieMock).when(activity).findViewById(R.id.sortie);

        doReturn(spChoixConversionMock).when(activity).findViewById(R.id.conversion);
        activity.findViews();
    }
}
```

On crée notre premier test afin de vérifier si la méthode `getEntree()` renvoie la bonne valeur. En fait on vérifie si l'élément sortie est le même que saisi initialement (vert).

```

43  @Test
44  public void getEntreeLitValeurSaisie()
45  {
46      // arrange : placer une chaîne représentant un nombre correct dans etEntreeMock
47      when(etEntreeMock.getText()).thenReturn(new FakeEditable( text: "12345.6789"));
48      // act : appeler getEntree()
49      double res = activity.getEntree();
50      // assert : vérifier que le résultat de getEntree() est le nombre voulu
51      assertThat( reason: "check if getEntree is ok", res, equalTo(Double.parseDouble(String.valueOf(etEntreeMock.getText()))));
52  }
53  }

```

De la même manière, on écrit le même programme mais on essaie de faire passer un autre type de valeur que du type *Double* (bleu). On ajoute une exception (rose) qui affiche l'erreur précise (violet).

```

54  @Test
55  public void getEntreeThrowNumberFormatException() throws Exception
56  {
57      // arrange : placer une chaîne incorrecte dans etEntreeMock
58      when(etEntreeMock.getText()).thenReturn(new FakeEditable( text: "7 nains"));
59      // act : appeler getEntree()
60      double res = activity.getEntree();
61      // pas d'assert mais un paramètre expected dans @Test
62  }
63  }
64  }
65  |

```

Tests failed: 1 of 1 test - 9s 843 ms

"C:\Program Files\Android\Android Studio\jre\bin\java.exe" ...

java.lang.NumberFormatException: For input string: "7 nains"

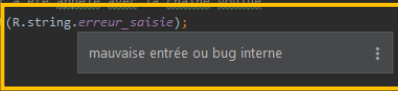
at sun.misc.FloatingDecimal.readJavaFormatString(FloatingDecimal.java:2043)
at sun.misc.FloatingDecimal.parseDouble(FloatingDecimal.java:110)
at java.lang.Double.parseDouble(Double.java:538)

On s'intéresse ensuite aux messages d'erreurs personnalisables. Ainsi quand on fait appel à la méthode `putSortieError()` et qu'on passe la souris sur le message d'erreur on obtient le texte personnalisé (orange).

```

68  @Test
69  public void putSortieEcritErreur()
70  {
71      // arrange : rien
72      // act : appeler putSortieError()
73      activity.putSortieError();
74      // assert : tvSortieMock.setText a été appelé avec la chaîne voulue
75      verify(tvSortieMock).setText(R.string.erreur_saisie);
76  }
77
78  }
79

```



Par la méthode suivante, on appelle `putSortie(nombre)` avec un nombre précis et on vérifie si c'est bien lui qui est affiché dans `tvSortieMock` (marron).

```

78  @Test
79  public void putSortieEcritValeurVoulue()
80  {
81      // arrange : rien
82      // act : appeler putSortie avec un nombre assez compliqué
83      activity.putSortie( nombre: 25.361);
84      // assert : tvSortieMock.setText a été appelé avec la chaîne voulue
85      verify(tvSortieMock).setText("25,361");
86  }
87

```

Enfin, on effectue les tests de conversion de psi en bar grâce à la méthode `convertirDoitConvertir()` qui vérifie si la conversion réalisée donne le bon résultat en utilisant la vue au clic sur le bouton de conversion (rouge).

```

88  public void convertirDoitConvertir()
89  {
90      // arrange : mettre en mode psi->bar
91      activity.getConvertisseur().setCodeConversion(1);
92      // arrange : placer -7.25 dans etEntreeMock
93      when(etEntreeMock.getText()).thenReturn(new FakeEditable( text: "-7.25"));
94
95      // act : appeler onConvertir(null)
96      activity.onConvertir( view: null);
97      // assert : tvSortieMock.setText a été appelé avec la chaîne voulue, "-0,500"
98      verify(tvSortieMock).setText("-0,500");
99  }
100

```

- Tests de l'interface

On passe maintenant à des tests automatiques sur AVD, avec Espresso. En fait, le but est de coder les actions qui s'effectueront automatiquement au lancement du test.

On crée donc une nouvelle classe. Dans cette dernière vous pouvez observer la présence de l'élément `@Rule` qui permet de récupérer l'objet Java représentant l'activité `MainActivity` sur l'AVD (rouge). Cette fois, les vues seront les véritables vues. C'est cet élément qui permet l'interaction avec l'application.

```
@RunWith(AndroidJUnit4.class)
public class TestsInterface
{
    @Rule
    public ActivityTestRule<MainActivity> mActivityRule =
        new ActivityTestRule<>(MainActivity.class);
}
```

On commence par récupérer sur la vue une valeur entrée lors du lancement du test (vert). On observe le résultat sur notre simulateur (bleu). C'est d'ailleurs assez impressionnant à voir la première fois puisque c'est l'application qui se charge d'écrire elle-même les valeurs !!

```

40  @Test
41  public void getEntreeLitValeurTapee()
42  {
43      // arrange : récupérer l'activité
44      final MainActivity activity = mActivityRule.getActivity();
45      // act : taper une valeur et appeler getEntree()
46      onView(withId(R.id.entree)).perform(typeText( stringToBeTyped: "6542.541"),
47                                          closeSoftKeyboard());
48      double entree = activity.getEntree();
49      // assert : vérifier que le résultat de getEntree() est le nombre tapé
50      assertThat( reason: "vérifier que le résultat de getEntree() est le nombre
51  }
```

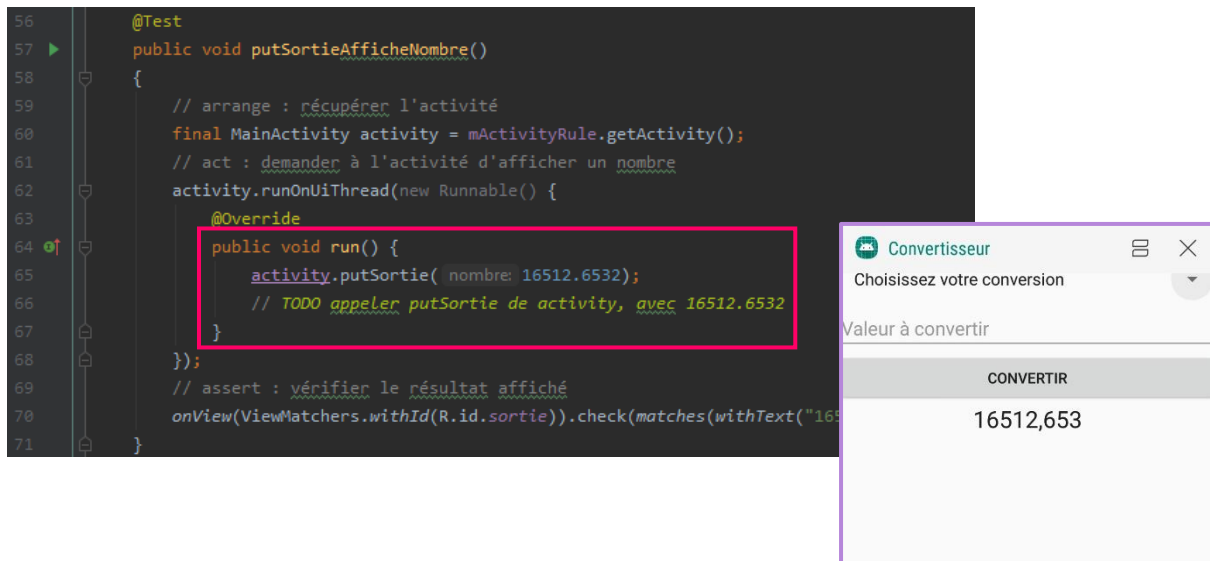
Convertisseur

Choisissez votre conversion ▼

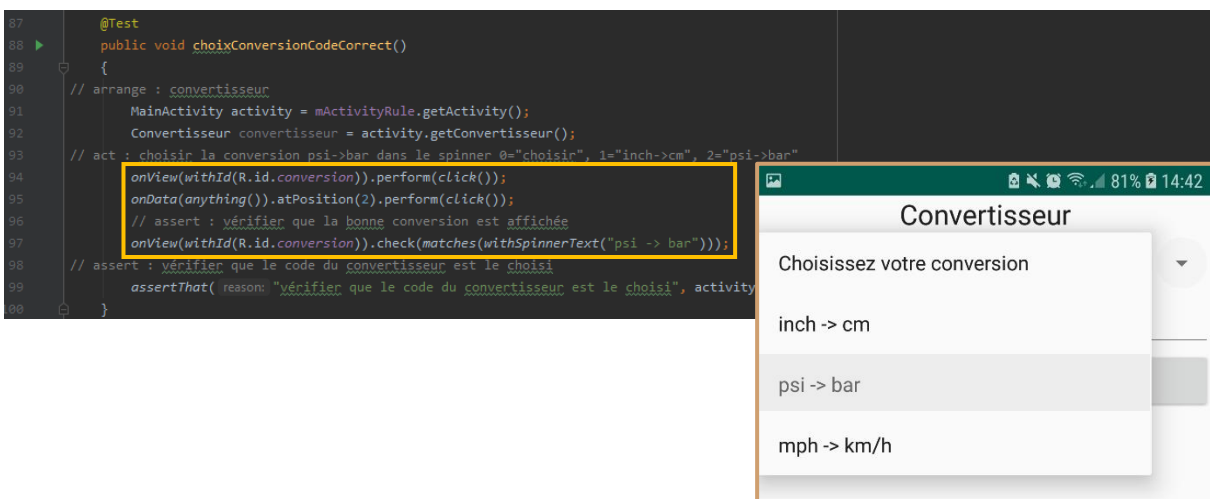
6542.541

CONVERTIR

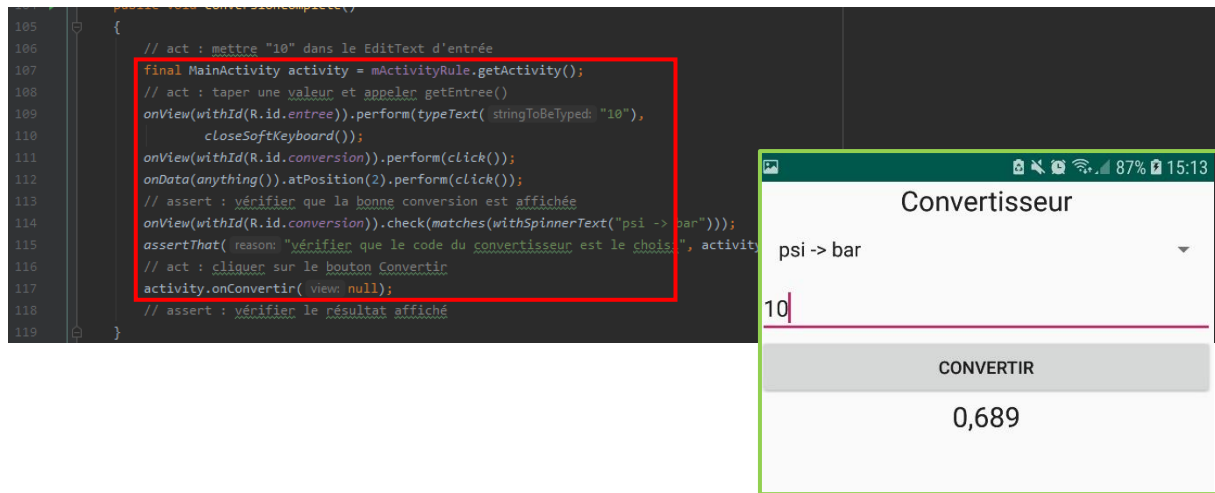
On vérifie ensuite que la sortie fonctionne correctement en suivant le même procédé (rose). Et on observe le résultat (violet).



On souhaite ensuite vérifier si la conversion choisie est bien celle sélectionnée (orange). On doit donc commander le clic sur le bouton et on observe le résultat (marron).



Enfin, on réalise le test de conversion en utilisant tout ce que nous avons utilisé précédemment. On observe donc les 3 étapes décrites juste avant se dérouler successivement grâce au code suivant (rouge). On observe le résultat (vert) et on est heureux parce que ça marche !!!!



Ressenti personnel

Au début ce td semblait difficile est très long à réaliser mais finalement il a été assez simple et sympa à réaliser. On se rend compte de la praticité des tests pour une application surtout quand celle-ci est importante.