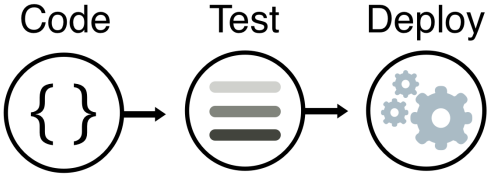




# Code Testing

## Using java



Marwén Saidi - marwen.saidi@orange.com

# JUnit 5

JUnit 5 est le framework de test le plus largement utilisé pour les applications Java. Depuis très longtemps, JUnit fait parfaitement son travail. Entre les deux, JDK 8 a apporté des fonctionnalités très intéressantes dans java et plus particulièrement les expressions lambda. JUnit 5 vise à adapter le style de codage java 8 et plusieurs autres fonctionnalités, c'est pourquoi java 8 est nécessaire pour créer et exécuter des tests dans JUnit 5 (bien qu'il soit possible d'exécuter des tests écrits avec JUnit 3 ou JUnit 4 pour la compatibilité descendante) .

## JUnit 5 Architecture

Par rapport à JUnit 4, JUnit 5 est composé de plusieurs modules différents issus de trois sous-projets différents:

JUnit 5 = JUnit Platform + JUnit Jupiter + JUnit Vintage

1. **JUnit Platform**
   Pour pouvoir lancer des tests junit, les IDE, les outils de construction ou les plugins doivent inclure et étendre les API de la plateforme. Il définit l'API TestEngine pour développer de nouveaux frameworks de test qui s'exécutent sur la plate-forme.
   Il fournit également un lanceur de console pour lancer la plate-forme à partir de la ligne de commande et créer des plugins pour Gradle et Maven.
2. **JUnit Jupiter**
   Il inclut de nouveaux modèles de programmation et d'extension pour l'écriture de tests. Il a toutes les nouvelles annotations junit et l'implémentation TestEngine pour exécuter des tests écrits avec ces annotations.
3. **JUnit Vintage**
   Son objectif principal est de prendre en charge l'exécution des tests écrits JUnit 3 et JUnit 4 sur la plate-forme JUnit 5. C'est une rétrocompatibilité.

## Installation

Vous pouvez utiliser JUnit 5 dans votre projet maven ou gradle en incluant au minimum deux dépendances, à savoir la dépendance de Jupiter Engine et la dépendance de Platform Runner.

```
//pom.xml

<properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <maven.compiler.source>1.8</maven.compiler.source>

<maven.compiler.target>${maven.compiler.source}</maven.compiler.target>
    <junit.jupiter.version>5.5.2</junit.jupiter.version>
    <junit.platform.version>1.5.2</junit.platform.version>
</properties>
<dependencies>
    <dependency>
        <groupId>org.junit.jupiter</groupId>
        <artifactId>junit-jupiter-engine</artifactId>
        <version>${junit.jupiter.version}</version>
        <scope>test</scope>
    </dependency>
    <dependency>
        <groupId>org.junit.platform</groupId>
        <artifactId>junit-platform-runner</artifactId>
        <version>${junit.platform.version}</version>
        <scope>test</scope>
    </dependency>
</dependencies>

//build.gradle

testRuntime("org.junit.jupiter:junit-jupiter-engine:5.5.2")
testRuntime("org.junit.platform:junit-platform-runner:1.5.2")
```

## JUnit 5 Annotations

JUnit 5 propose les annotations suivantes pour écrire des tests.

| Annotation | Description |
|---|---|
| @BeforeEach | La méthode annotée sera exécutée avant chaque méthode de test de la classe de test. |

| | |
|---|---|
| @AfterEach | La méthode annotée sera exécutée après chaque méthode de test dans la classe de test. |
| @BeforeAll | La méthode annotée sera exécutée avant toutes les méthodes de test de la classe de test. Cette méthode doit être statique. |
| @AfterAll | La méthode annotée sera exécutée après toutes les méthodes de test de la classe de test. Cette méthode doit être statique. |
| @Test | Il est utilisé pour marquer une méthode comme test junit |
| @DisplayName | Utilisé pour fournir un nom d'affichage personnalisé pour une classe de test ou une méthode de test |
| @Disable | Il est utilisé pour désactiver ou ignorer une classe ou une méthode de test de la suite de tests. |
| @Nested | Utilisé pour créer des classes de test imbriquées |
| @Tag | Mark teste des méthodes ou des classes de test avec des balises pour tester la découverte et le filtrage |
| @TestFactory | Mark a method est une usine de test pour les tests dynamiques |

## Ecrire des Tests de Junit 5

Il n'y a pas beaucoup de changement entre JUnit 4 et JUnit 5 dans les styles d'écriture de test. Voici des exemples de tests avec leurs méthodes de cycle de vie.

```
import org.junit.jupiter.api.AfterAll;
import org.junit.jupiter.api.AfterEach;
import org.junit.jupiter.api.Assertions;
import org.junit.jupiter.api.BeforeAll;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Disabled;
import org.junit.jupiter.api.Tag;
import org.junit.jupiter.api.Test;

import com.howtodoinjava.junit5.examples.Calculator;

public class AppTest {

    @BeforeAll
    static void setup(){
        System.out.println("@BeforeAll executed");
    }
```

```
    @BeforeEach
    void setupThis(){
        System.out.println("@BeforeEach executed");
    }

    @Tag("DEV")
    @Test
    void testCalcOne()
    {
        System.out.println("======TEST ONE EXECUTED=======");
        Assertions.assertEquals( 4 , Calculator.add(2, 2));
    }

    @Tag("PROD")
    @Disabled
    @Test
    void testCalcTwo()
    {
        System.out.println("======TEST TWO EXECUTED=======");
        Assertions.assertEquals( 6 , Calculator.add(2, 4));
    }

    @AfterEach
    void tearThis(){
        System.out.println("@AfterEach executed");
    }

    @AfterAll
    static void tear(){
        System.out.println("@AfterAll executed");
    }
}
```

## Test Suites

En utilisant les suites de tests JUnit 5, vous pouvez exécuter des tests répartis dans plusieurs classes de test et différents packages. JUnit 5 fournit deux annotations: @SelectPackages et @SelectClasses pour créer des suites de tests.

Pour exécuter la suite, vous utiliserez @RunWith (JUnitPlatform.class).

```
@RunWith(JUnitPlatform.class)
@SelectPackages("com.howtodoinjava.junit5.examples")
```

```
public class JUnit5TestSuiteExample
{
}
```

De plus, vous pouvez utiliser les annotations suivantes pour filtrer les packages de test, les classes ou même les méthodes de test.

- @IncludePackages et @ExcludePackages pour filtrer les packages
- @IncludeClassNamePatterns et @ExcludeClassNamePatterns pour filtrer les classes de test
- @IncludeTags et @ExcludeTags pour filtrer les méthodes de test

```
@RunWith(JUnitPlatform.class)
@SelectPackages("com.howtodoinjava.junit5.examples")
@IncludePackages("com.howtodoinjava.junit5.examples.packageC")
@ExcludeTags("PROD")
public class JUnit5TestSuiteExample
{
}
```

## Assertions

Les assertions aident à valider la sortie attendue avec la sortie réelle d'un cas de test. Pour simplifier les choses, toutes les assertions JUnit Jupiter sont des méthodes statiques de la classe org.junit.jupiter.Assertions, par exemple assertEquals (), assertNotEquals ().

```
void testCase()
{
    //Test will pass
    Assertions.assertNotEquals(3, Calculator.add(2, 2));

    //Test will fail
    Assertions.assertNotEquals(4, Calculator.add(2, 2),
"Calculator.add(2, 2) test failed");

    //Test will fail
    Supplier<String> messageSupplier  = ()-> "Calculator.add(2, 2) test
failed";
    Assertions.assertNotEquals(4, Calculator.add(2, 2),
messageSupplier);
}
```

## Assumptions

La classe Assumptions fournit des méthodes statiques pour prendre en charge l'exécution de tests conditionnels basés sur des hypothèses. Une hypothèse échouée entraîne l'arrêt d'un test. Les hypothèses sont généralement utilisées chaque fois que la poursuite de l'exécution d'une méthode de test donnée n'a pas de sens. Dans le rapport de test, ces tests seront marqués comme réussis.

La classe JUnit jupiter Assumptions a deux méthodes de ce type: assumeFalse (), assumeTrue ().

```
public class AppTest {
    @Test
    void testOnDev()
    {
        System.setProperty("ENV", "DEV");
        Assumptions.assumeTrue("DEV".equals(System.getProperty("ENV")),
AppTest::message);
    }

    @Test
    void testOnProd()
    {
        System.setProperty("ENV", "PROD");

Assumptions.assumeFalse("DEV".equals(System.getProperty("ENV")));
    }

    private static String message () {
        return "TEST Execution Failed :: ";
    }
}
```

## Backward Compatibility for JUnit 3 or JUnit 4

JUnit 4 existe depuis assez longtemps et il existe de nombreux tests écrits en junit 4. JUnit Jupiter doit également prendre en charge ces tests. Pour cela, un sous-projet JUnit Vintage est développé.

JUnit Vintage fournit une implémentation TestEngine pour exécuter des tests basés sur JUnit 3 et JUnit 4 sur la plate-forme JUnit 5.

# Mockito 2

## Mockito Introduction

Lors des tests unitaires de l'application, il n'est parfois pas possible de répliquer exactement l'environnement de production. Parfois, la base de données n'est pas disponible et parfois l'accès au réseau n'est pas autorisé. Il peut y avoir de nombreuses autres restrictions de ce type. Pour faire face à de telles limitations, nous devons créer des simulations pour ces ressources indisponibles.

Mockito est un framework open source qui vous permet de créer facilement des doubles de test (mocks). Test Double est un terme générique pour tous les cas où vous remplacez un objet de production à des fins de test. Dans mockito, nous travaillons généralement avec les types de doubles de test suivants.

- Stubs - est un objet qui a des valeurs de retour prédéfinies pour les exécutions de méthode effectuées pendant le test.
- Spies - sont des objets similaires aux stubs, mais qui enregistrent en plus la manière dont ils ont été exécutés.
- Mocks - sont des objets qui ont des valeurs de retour aux exécutions de méthode effectuées pendant le test et qui ont enregistré les attentes de ces exécutions. Les simulacres peuvent lever une exception s'ils reçoivent un appel auquel ils ne s'attendent pas et sont vérifiés lors de la vérification pour s'assurer qu'ils ont reçu tous les appels qu'ils attendaient.

Nous pouvons nous moquer à la fois des interfaces et des classes dans la classe de test. Mockito aide également à produire un code passe-partout minimum tout en utilisant des annotations mockito.

## Mockito Setup

Pour ajouter mockito dans le projet, nous pouvons ajouter la version mockito souhaitée par n'importe quel moyen, c'est-à-dire un fichier maven, gradle ou jar.

```
<dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito-core</artifactId>
    <version>2.23.4</version>
    <scope>test</scope>
</dependency>
```

```
testCompile group: 'org.mockito', name: 'mockito-core', version:
'2.23.4'
```

Le fichier pom complet ayant toutes les dépendances pour créer des tests Junit avec mockito est donné ci-dessous. Sinon, nous devons rechercher et ajouter des versions correspondantes de `objenesis, hamcrest-core, byte-buddy, byte-buddy-agent, junit, mockito.`

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.howtodoinjava.demo</groupId>
    <artifactId>MockitoExample</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <dependencies>
        <dependency>
            <groupId>org.mockito</groupId>
            <artifactId>mockito-core</artifactId>
            <version>2.23.0</version>
            <scope>test</scope>
        </dependency>
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.12</version>
            <scope>test</scope>
        </dependency>
    </dependencies>
</project>
```

# Mockito Annotations

Avant de prendre le clavier pour écrire des tests d'application et des tests unitaires, passons rapidement en revue les annotations factices utiles.

- **@Mock** est utilisé pour la création de simulation. Cela rend la classe de test plus lisible.
- **@Spy** est utilisé pour créer une instance d'espionnage. Nous pouvons l'utiliser à la place de la méthode spy (Object).
- **@InjectMocks** est utilisé pour instancier automatiquement l'objet testé et y injecter toutes les dépendances de champs annotés @Mock ou @Spy (le cas échéant).
- **@Captor** est utilisé pour créer un capteur d'argument

Pour traiter toutes les annotations ci-dessus, **MockitoAnnotations.initMocks (testClass)**; doit être utilisé au moins une fois. Pour traiter les annotations, nous pouvons utiliser le runner intégré **MockitoJUnitRunner** ou la règle **MockitoRule**. Nous pouvons également invoquer explicitement la méthode `initMocks()` dans la méthode Junit annotée `@Before`.

//1

```
@RunWith(MockitoJUnitRunner.class)
public class ApplicationTest {
    //code
}
```

//2

```
public class ApplicationTest {
    @Rule public MockitoRule rule =
MockitoJUnit.rule().strictness(Strictness.STRICT_STUBS);

    //code
}
```

//3

```
public class ApplicationTest {
    @Before
    public void init() {
        MockitoAnnotations.initMocks(this);
    }
}
```

# JUnit Mockito Exemple

Apprenons à écrire des tests junit qui utilisent des simulations pour les dépendances. L'exemple donné a un **RecordService** qui stocke un fichier donné dans la base de données et un emplacement réseau à l'aide de **DatabaseDAO** et **NetworkDAO**.

Dans l'environnement de test, il n'est pas possible d'accéder à la base de données ou à l'emplacement réseau, nous créons donc des simulations pour les deux référentiels.

```
public class DatabaseDAO
{
    public void save(String fileName) {
        System.out.println("Saved in database");
    }
}
```

```
public class NetworkDAO
{
    public void save(String fileName) {
        System.out.println("Saved in network location");
    }
}
```

```
public class RecordService
{
    DatabaseDAO database;
    NetworkDAO network;

    //setters and getters

    public boolean save(String fileName)
    {
        database.save(fileName);
        System.out.println("Saved in database in Main class");

        network.save(fileName);
```

```
        System.out.println("Saved in network in Main class");

        return true;
    }
}
```

Veuillez garder à l'esprit que si nous utilisons un framework DI tel que Spring, nous aurions peut-être utilisé l'annotation **@Autowired**.

Pour tester cette classe RecordService, créons un test unitaire.

```
import static org.junit.Assert.assertEquals;
import static org.mockito.Mockito.times;
import static org.mockito.Mockito.verify;

import org.junit.Test;
import org.junit.runner.RunWith;
import org.mockito.InjectMocks;
import org.mockito.Mock;
import org.mockito.junit.MockitoJUnitRunner;

import com.howtodoinjava.demo.mockito.DatabaseDAO;
import com.howtodoinjava.demo.mockito.NetworkDAO;
import com.howtodoinjava.demo.mockito.RecordService;

@RunWith(MockitoJUnitRunner.class)
public class ApplicationTest
{
    @InjectMocks
    RecordService recordService;

    @Mock
    DatabaseDAO databaseMock;

    @Mock
    NetworkDAO networkMock;

    @Test
    public void saveTest()
    {
        boolean saved = recordService.save("temp.txt");
        assertEquals(true, saved);
```

```
        verify(databaseMock, times(1)).save("temp.txt");
        verify(networkMock, times(1)).save("temp.txt");
    }
}
```

Voyons ci-dessus la classe de test étape par étape:

- Annotez le test avec @RunWith (MockitoJUnitRunner.class) afin que mockito puisse traiter les annotations.
- Annotez les champs DAO avec l'annotation @Mock pour avoir un objet fictif instancié pour les deux.
- Annotez le champ de service avec l'annotation @InjectMocks pour d'abord instancier puis injecter les deux dépendances simulées.
- Appelez la méthode à tester sur la classe à tester ie. recordService.
- Vérifiez que les méthodes des objets simulés ont été appelées.

# Mockito annotations – @Mock, @Spy, @Captor, @InjectMocks

## Mockito Annotations

### @Mock

L'annotation **@Mock** est utilisée pour créer et injecter des instances simulées. Nous ne créons pas d'objets réels, nous demandons plutôt à mockito de créer une maquette pour la classe.

L'annotation @Mock est une alternative à Mockito.mock (classToMock). Ils obtiennent tous les deux le même résultat. L'utilisation de @Mock est généralement considérée comme «plus propre», car nous ne remplissons pas les tests avec des affectations standard qui se ressemblent toutes.

Utilisation de l'annotation @Mock

- permet la création abrégée des objets requis pour les tests.
- minimise le code de création de simulation répétitif.

- rend la classe de test plus lisible.
- rend la vérification d'erreur plus facile à lire car le nom du champ est utilisé pour identifier la maquette.

Dans l'exemple donné, nous nous sommes moqués de la classe HashMap. Dans les tests réels, nous nous moquerons des classes d'application réelles. Nous avons placé une paire clé-valeur dans la carte, puis avons vérifié que l'appel de méthode était effectué sur une instance de carte simulée.

```
@Mock
HashMap<String, Integer> mockHashMap;

@Test
public void saveTest()
{
    mockHashMap.put("A", 1);

    Mockito.verify(mockHashMap, times(1)).put("A", 1);
    Mockito.verify(mockHashMap, times(0)).get("A");

    assertEquals(0, mockHashMap.size());
}
```

### @Spy

L'annotation @Spy est utilisée pour créer un objet réel et espionner cet objet réel. Un espion aide à appeler toutes les méthodes normales de l'objet tout en suivant chaque interaction, comme nous le ferions avec un simulacre.

Notez dans l'exemple donné comment la taille de la carte est maintenue à 1 car nous y avons ajouté une paire clé-valeur. Nous sommes également en mesure de récupérer la valeur ajoutée de la carte en utilisant sa clé. Ce n'est pas possible dans les instances simulées.

```
@Spy
HashMap<String, Integer> hashMap;

@Test
public void saveTest()
{
    hashMap.put("A", 10);

    Mockito.verify(hashMap, times(1)).put("A", 10);
```

```
    Mockito.verify(hashMap, times(0)).get("A");

    assertEquals(1, hashMap.size());
    assertEquals(new Integer(10), (Integer) hashMap.get("A"));
}
```

### La différence entre @Mock et @Spy

Lors de l'utilisation de @Mock, mockito crée une instance de shell simple de la classe, entièrement instrumentée pour suivre les interactions avec elle. Ce n'est pas un objet réel et ne maintient pas les changements d'état de celui-ci.

Lorsque vous utilisez @Spy, mockito crée une instance réelle de la classe et suit toutes les interactions avec elle. Il maintient l'état des changements.

## @Captor

L'annotation @Captor est utilisée pour créer une instance **ArgumentCaptor** qui est utilisée pour capturer les valeurs d'argument de méthode pour d'autres assertions.

Notez que mockito vérifie les valeurs des arguments en utilisant la méthode equals() de la classe d'arguments.

```
@Mock
HashMap<String, Integer> hashMap;

@Captor
ArgumentCaptor<String> keyCaptor;

@Captor
ArgumentCaptor<Integer> valueCaptor;

@Test
public void saveTest()
{
    hashMap.put("A", 10);

    Mockito.verify(hashMap).put(keyCaptor.capture(),
valueCaptor.capture());

    assertEquals("A", keyCaptor.getValue());
    assertEquals(new Integer(10), valueCaptor.getValue());
}
```

### @InjectMock

Dans mockito, nous devons créer l'objet de classe à tester et insérer ses dépendances (moquées) pour tester complètement le comportement. Pour ce faire, nous utilisons l'annotation @InjectMocks.

@InjectMocks marque un champ sur lequel l'injection doit être effectuée. Mockito essaiera d'injecter des simulations uniquement par injection de constructeur, injection de setter ou injection de propriété - dans cet ordre. Si l'une des stratégies d'injection donnée échoue, Mockito ne signalera pas d'échec.

## Initialisation des annotations mockito

Pour utiliser les annotations ci-dessus, la classe de test doit initialiser l'annotation en utilisant l'une des trois méthodes suivantes:

1. Utilisez @RunWith (MockitoJUnitRunner.class) en haut de la classe de test unitaire.

```
@RunWith(MockitoJUnitRunner.class)
public class ExampleTest {

    @Mock
    private List list;

    @Test
    public void shouldDoSomething() {
        list.add(100);
    }
}
```

2. Utilisez MockitoAnnotations.initMocks (this) dans la méthode @Before de la classe de test unitaire.

```
public class ExampleTest {

      @Mock
      private List list;

      @Before
      public void setup() {
      MockitoAnnotations.initMocks(this);
      }
```

```
      @Test
      public void shouldDoSomething() {
      list.add(100);
      }
}
```

3. Utilisez MockitoJUnit.rule () pour créer la classe MockitoRule.

```
public class ExampleTest {

    @Rule
    public MockitoRule rule = MockitoJUnit.rule();

    @Mock
    private List list;

    @Test
    public void shouldDoSomething() {
        list.add(100);
    }
}
```