Mockito pour Java

`http://mockito.org/`
`https://static.javadoc.io/org.mockito/mockito-core/2.13.0/org/mockito/Mockito.html#1`

Mockito est un générateur automatique de doublures qui s'utilise ainsi dans un test :
- dans la phase de création des objets du test, on crée les mocks ;
- toujours dans cette phase, on décrit leur comportement (phase de *stubbing*) ;
- lors de l'exécution du code à tester, toutes les interactions avec les mocks sont mémorisées par Mockito ;
- l'oracle peut interroger les mocks pour savoir comment ils ont été utilisés.

```
import static org.mockito.Mockito.*;
```

Création d'un mock `Carte` est une interface (SVL) ou une classe.
`Carte carte = mock(Carte.class);` le mock n'est pas nommé dans les messages d'erreurs, ou
`Carte carte = mock("macarte", Carte.class);` le mock est nommé `macarte` ou
`@Mock Carte carte` : le nom du mock sera `carte`.
Le mock est en l'état capable d'accepter tous les appels de méthode de l'interface, en retournant des valeurs par défaut (0 pour un type numérique, faux pour un type booléen, null pour un type objet, etc).

Stubbing, dire au mock comment se comporter
- retour d'une valeur : `when(carte.solde()).thenReturn(56);`
- levée d'exception : `doThrow` resp. `thenThrow` pour une méthode de type `void` resp. non `void`
  `doThrow(new SoldeInsuffisantException()).when(carte).debiter(50);`
  `when(obj.foo().thenThrow(new NullPointerException());`

Ces comportements écrasent les comportements par défaut. En l'état tous les appels à `carte.solde()` retournent 56, mais un `when` ultérieur écraserait le 56. Si un paramètre est fourni à la méthode stubbée, alors le comportement ne vaut que pour ce paramètre : si `debiter(50)` est appelé sur `carte` il y a levée de `SoldeInsuffisantException` mais pour `debiter(2)` le comportement sera celui par défaut.
Comportement non utilisé par le test : si le comportement décrit dans la phase de stubbing n'est pas appelé lors de l'exécution du test, Mockito ne se plaint pas.
Stubbing avancé : Il est possible d'enchaîner les `thenReturn` :
avec `when(carte.solde()).thenReturn(56).thenReturn(3);` le premier appel à `carte.solde()` retournera 56 et tous les suivants retourneront 3. Il est aussi possible d'enchaîner des `thenReturn` et des `thenThrow`, le `thenThrow` prenant en paramètre l'exception levée, comme le `doThrow`.

Vérification que les interactions avec le mock sont celles attendues

`verify(carte).debiter(50);` lève une exception si la méthode `debiter` n'a pas été appelée exactement une fois avec `50` en paramètre.
Vérifications avancées :
- `verify(mock, times(2)).methode();` (le défaut est `times(1)`)
- `verify(mock, atLeastOnce()).methode();`
- `verify(mock, atMost(1)).methode();`
- `verify(mock, never()).methode();`
- `verifyNoMoreInteractions(mock);`
- `verifyZeroInteractions(mock);`

On peut aussi vérifier l'ordre dans lequel les appels ont eu lieu : `org.mockito.InOrder`.

Utilisation avancée : Argument Matchers

Les *argument matchers* permettent de ne pas préciser une valeur d'argument, ce qui sert à la fois en phase de *stubbing* et de vérification : `any()` (tout objet ou null), `anyInt()` (tout entier, Integer ou null), `anyString()`, etc.

```
verify(carte, never()).debiter(anyInt());
doThrow(new SoldeInsuffisantException()).when(carte).debiter(anyInt());
```