Mockito

# ¿Qué es Mockito?

Mocking es una forma de probar la funcionalidad de una clase de forma aislada. Mocking no requiere una conexión a la base de datos o la lectura de un archivo de propiedades o de un servidor de archivos para probar una funcionalidad. Los objetos mock hacen el mock del servicio real. Un objeto mock devuelve un dato ficticio correspondiente a una entrada ficticia que se le pasa.

Mockito facilita la creación de objetos simulados sin problemas. Utiliza Java Reflection para crear objetos simulados para una interfaz determinada. Los objetos simulados no son más que un proxy de las implementaciones reales.

# Estilo BDD

El término BDD (Behavior Driven Development) fue definido por Dan North en 2006 y fomenta la escritura de pruebas en un lenguaje natural y legible para el ser humano que se centra en el comportamiento de la aplicación.

Define una forma claramente estructurada de escribir pruebas siguiendo tres secciones (Arrange, Act, Assert):

* dadas algunas condiciones previas (Arrange)
* cuando se produce una acción (Act)
* entonces verifica la salida (Assert)

# Implementando Mockito.

A continuación se muestra como implementar Mockito con Junit versión 5 en un proyecto maven.

## Indicando dependencias

Asumiendo que ya se tienen indicadas las dependencias de Junit versión 5, para mockito agregamos dentro de nuestro proyecto Maven dos dependencias:

1. Librería principal de mockito:

*<dependency>*

*<groupId>org.mockito</groupId>*

*<artifactId>mockito-core</artifactId>*

*<version>2.21.0</version>*

*<scope>test</scope>*

*</dependency>*

1. Librería donde se incluye el modulo de mockito para junit para utilizado como extensión del mismo:

*<dependency>*

*<groupId>org.mockito</groupId>*

*<artifactId>mockito-junit-jupiter</artifactId>*

*<version>2.23.0</version>*

*<scope>test</scope>*

*</dependency>*

**Nota**: En caso de no tener indicado el plugin de surefire para reporte y ejecución de los tests, colocamos la siguiente especificación:

*<plugin>*

*<artifactId>maven-surefire-plugin</artifactId>*

*<version>2.19.1</version>*

*<dependencies>*

*<dependency>*

*<groupId>org.junit.platform</groupId>*

*<artifactId>junit-platform-surefire-provider</artifactId>*

*<version>1.0.1</version>*

*</dependency>*

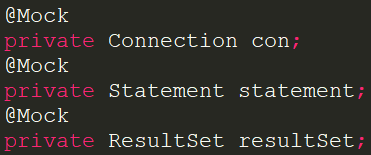
*</dependencies>*

*</plugin>*

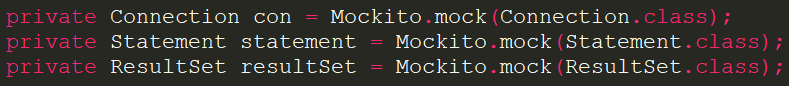
## Creando los mocks

Los Mocks son simulaciones de objetos en los cuales que podamos manipular o camuflar los retornos de los comportamientos deseados. Para declararlos tenemos 2 formas:

1. Anotación.



1. Manual.



A partir de estas simulaciones obtenemos un “cascaron” del objeto, es decir podremos acceder e invocar sus métodos, pero estos no ejecutarán ningún comportamiento o lógica y retornarán un nulo (null). Para indicarle a un Mock como queremos que se componte hay dos formas básicas:

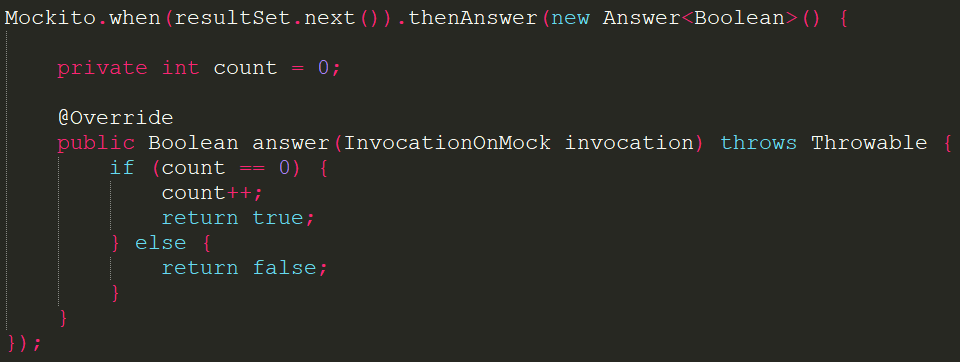
1. Indica el resultado ESTATICO de un comportamiento dado:



1. Dado un comportamiento x, indica este resultado ESTATICO.



En caso de que queramos indicar como resultado un comportamiento condicionado o dinámico, debemos hacer uso de la interfaz Answer, donde en esta simplemente sobreescribiendo su método “answer” podemos indicar que dato queremos como resultado del mock. Ejemplo:

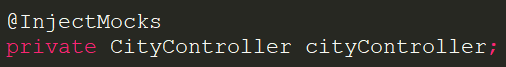


En este ejemplo, mediante los métodos answer de la interfaz indicamos que retornamos un booleano en función a un contador.

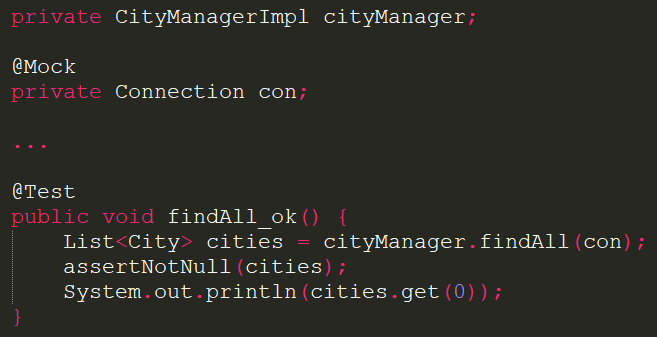
## Inyectando Mocks

El objetivo final de realizar mocks es simular comportamientos de ciertos elementos que en el momento no tenemos a mano. Esto involucra necesitar probar un objeto que invoca a otros objetos y estos otros objetos no estar disponibles, como por ejemplo una base de datos o un servicio web. En estos casos, Mockito nos ofrece la posibilidad de hacer mocks de estos elementos e “inyectarlos” para ser aprovechados por los diferentes casos de prueba del elemento en cuestión.

Para ello se auxilia de la anotación @InyectMocks y a continuación el objeto al cual le queremos inyectar estos, en caso de este recibir los objetos por inyección de dependencias.



En caso de que los objetos simulados (Mocks) no sean inyectamos, podemos pasarlos de forma directa por el medio que la clase disponga, como por ejemplo en forma de parámetro al método a probar o bien asociándolo a alguna propiedad.



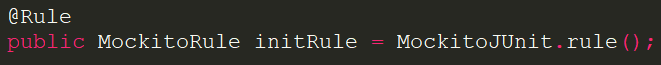
## Habilitando Mocks

Luego de tener todos nuestros mocks creados y especificados los retornos de los métodos que queremos controlar, debemos indicar de qué forma habilitar Mockito y todos los elementos utilizados. Para ello existen 4 formas básicas:

1. Invocando el método initMocks(): generalmente se hace en el método setUp de la clase de prueba, pero puede hacerse al inicio de cada caso de test. Lo importante es habilitarlo antes de ejecutar el caso de prueba.



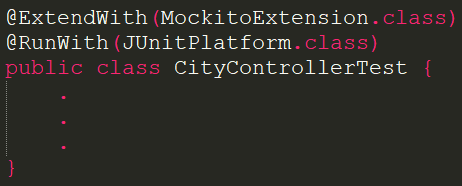
1. Creando un MockitoRule: la creación de este elemento automáticamente habilita todos los mocks declarados dentro del proceso de prueba unitaria.



1. Indicando el Runner de Mockito como runner principal de la clase de prueba:



1. Extensión de Junit: Para indicar que se va a implementar Mockito como extensión de JUnit debemos hacer uso la anotación @ExtendWith, a la cual le indicaremos el extensor de mockito y, a partir de ahí, podremos hacer uso de elementos de Mockito que nos ayudaran en nuestras pruebas.



# Uso de Spy

Mockito proporciona la opción de crear un espía en los objetos reales. Cuando se llama al espía, se llama al método real del objeto real mediante la creación de un Mock Parcial. Este a su vez permite también poder hacer mocks de métodos de este objeto, obteniendo lo mejor de ambas partes. Tenemos dos formas de utilizarlo:

1. Anotación



1. Manual



# Referencias

* <https://site.mockito.org/>
* <https://www.baeldung.com/mockito-series>
* <https://www.tutorialspoint.com/mockito/index.htm>
* <https://mkyong.com/tag/mockito/>