How Mockito Works

How Mockito works?

Mockito works by storing and retrieving method invocation details on mocks using method interception.
Method interception is a technique used in Aspect Oriented Programming(AOP) to address cross-cutting concerns *in a software system like logging, statistics etc., by intercepting method calls. I find this short description from Google Guice helpful if you are unfamiliar*

Mockito uses the runtime code generation and manipulation library Byte Buddy for interception and the hassle-free reflection library Objenesis for initializing mock objects

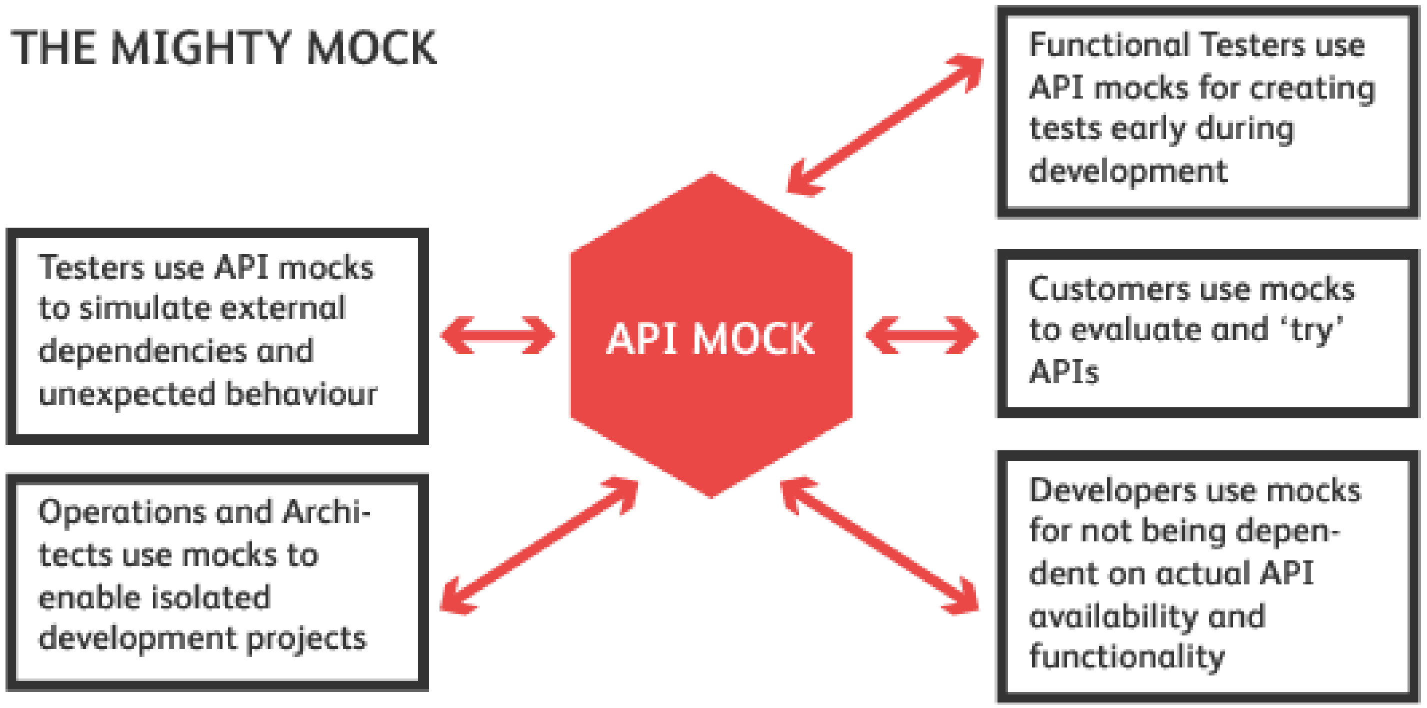
How Mockito works?

There are two types of mocking frameworks **- Proxy-based and Classloader remapping-based frameworks.**

- In Proxy-based mocking, a proxy object imitates the actual object. We inject the proxy object as a dependency through either the constructor or the setter.
- Dependency Injection (DI) frameworks such as Spring utilize this form of mocking. It is also used by frameworks such as *EasyMock*, *JMock*, and *Mockito*.

Classloader remapping-based frameworks

In classloader remapping-based mocking, the `.class` file of a dependency is remapped to the `.class` file of the mock object by the class-loader. Therefore, when a dependency is required the mock object is loaded instead of the actual object. Mocking frameworks such as *JMockit* and *PowerMock* support this form of mocking. Classloader remapping-based mocking can mock static/private/final methods or final classes, unlike proxy-based mocking.


THE MIGHTY MOCK
API MOCK
Functional Testers use API mocks for creating tests early during development
Testers use API mocks to simulate external dependencies and unexpected behaviour
Customers use mocks to evaluate and 'try' APIs
Operations and Archi-tects use mocks to enable isolated development projects
Developers use mocks for not being depen-dent on actual API availability and functionality

https://devopedia.org/mock-testing

Advantages and Disadvantages

Mockito supports the creation of mock objects and spies. Mocking is a powerful concept in testing across languages. Without mocking:

- It will take much boilerplate code to set up some system dependencies (web servers, databases, and services that require calls made over the internet) for testing our system.
- The execution of the test suite will also be slow.
- It will be impossible to test error conditions, exceptions, and functions that perform time-consuming tasks such as deleting files.

Disadvantages of mocks include:

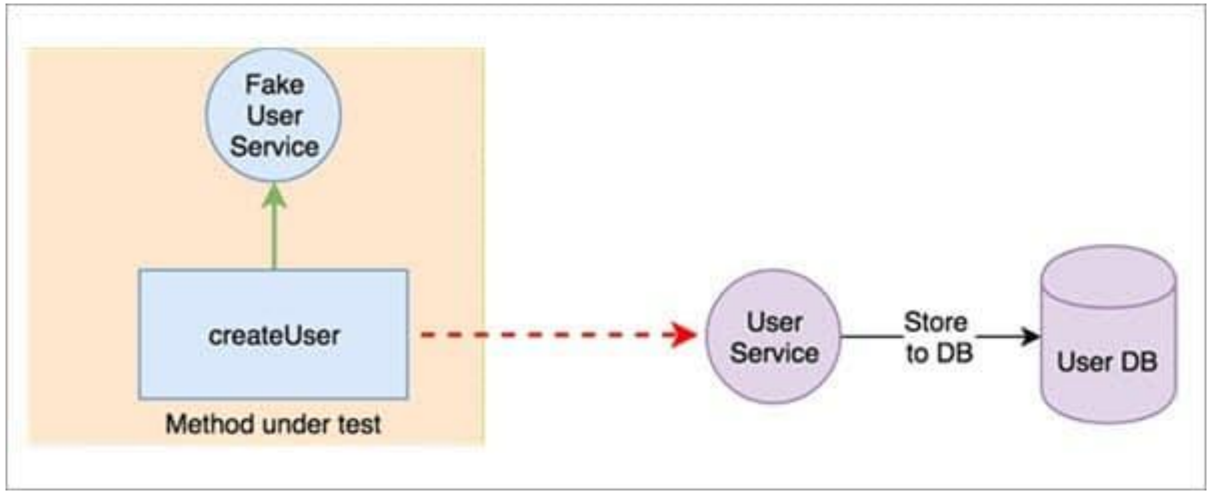
- Mocking systems make our test classes run slowly since they depend on reflection
- Over mocking leads to over-abstraction. This makes our code extremely complicated.

Robert C. Martin suggests mocking sparingly. He recommends that we "mock across architecturally significant boundaries, but not within those boundaries."

- He means that we should only mock out the database, web servers, and any other external service.
- For example, parts of our code that interact with SMTP servers to send emails or perform I/O operations.

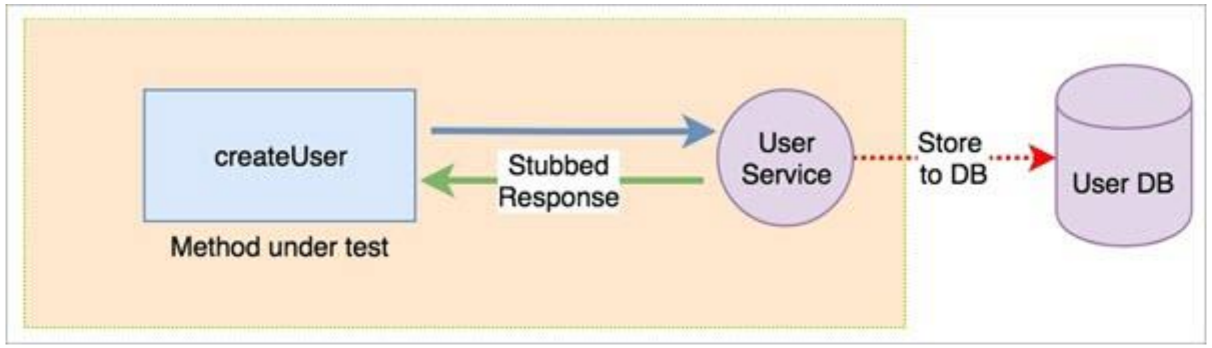
Fakes

- A fake is a working implementation similar to a real dependency, except the fact that it is local to the system under test.
- **Example:** Instead of hitting a real production DB, the test uses a simple collection/in-memory to store data.



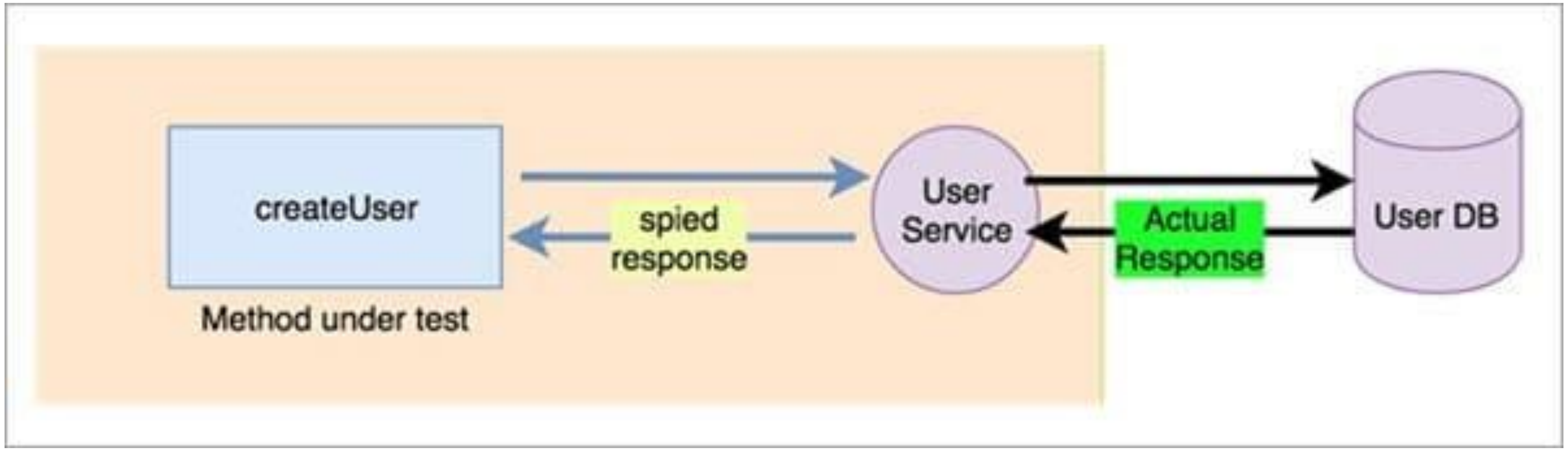
Stubs

- Stubs are pre-configured responses when a dependency is called from the system under test.



Spies

- As the name suggests, its actually the real function (dependency) call with some watching mechanism. Post the call, it can be verified whether the call was actually triggered or not along with the parameters



Mocks

- Mocks are special instances of objects, on which Stubbed/pre-configured responses can be specified. The fact that the mock got called can be verified as an assert in the test.

Mocks and Stubs

- Mocks and stubs are fake Java classes that replace these external dependencies. These fake classes are then instructed before the test starts to behave as you expect. More specifically:
- A stub is a fake class that comes with preprogrammed return values. It's injected into the class under test to give you absolute control over what's being tested as input. A typical stub is a database connection that allows you to mimic any scenario without having a real database.
- A mock is a fake class that can be examined after the test is finished for its interactions with the class under test. For example, you can ask it whether a method was called or how many times it was called. Typical mocks are classes with side effects that need to be examined, e.g. a class that sends emails or sends data to another external service.

What Is a Spy?

- A spy is the other type of test double that Mockito creates. In contrast to creating a mock, creating a spy requires an instance to spy on. By default, a spy delegates all method calls to the real object and records what method was called and with what parameters. That’s what makes it a spy: It’s spying on a real object.

Default Return Values

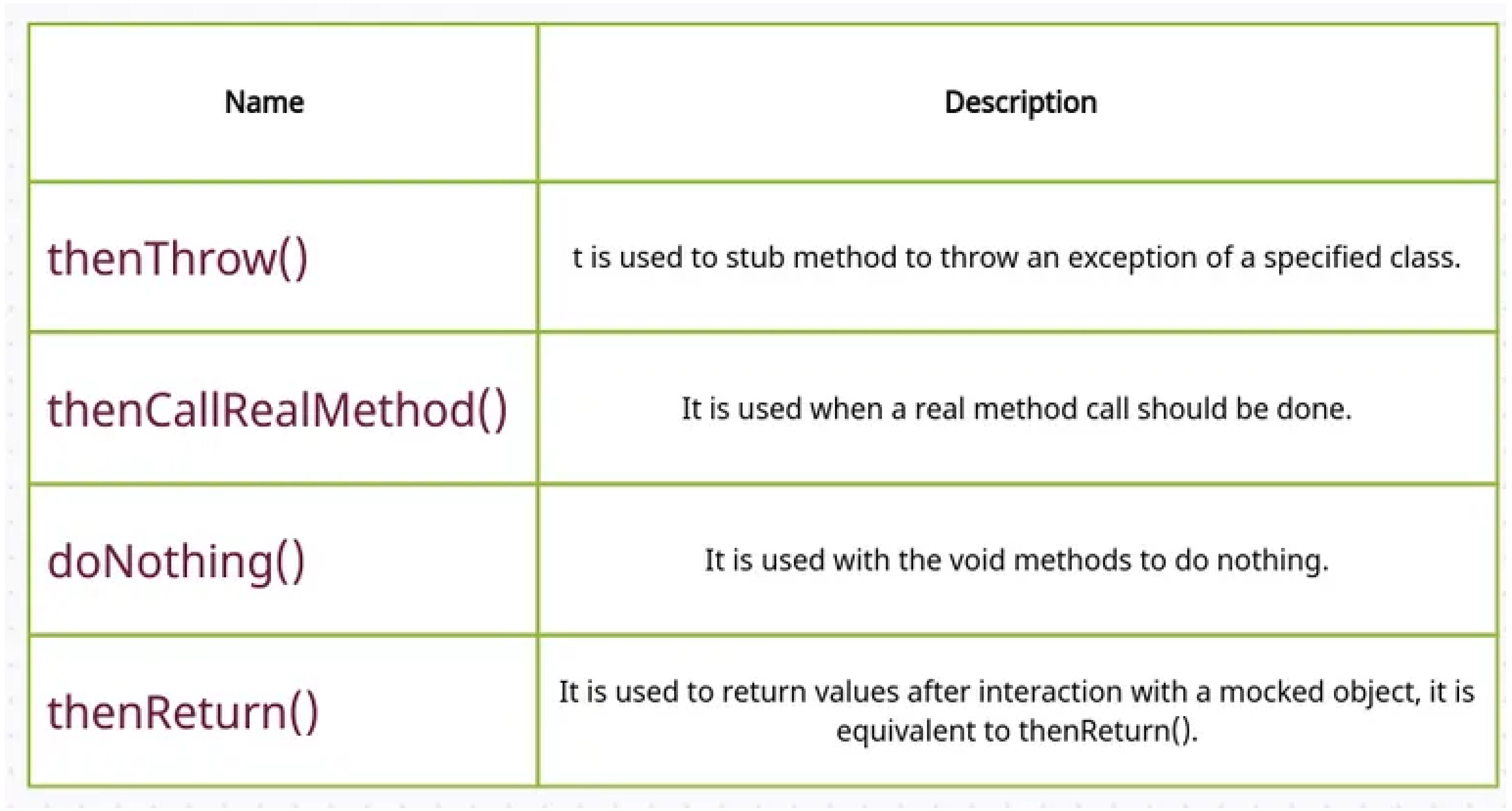
- Consider the following Interface

```
interface Demo {
int getInt();
Integer getInteger();
double getDouble();
boolean getBoolean();
String getObject();
Collection<String>
getCollection();
String[] getArray();
Stream<?> getStream();
Optional<?> getOptional();
 }
```

Default Example

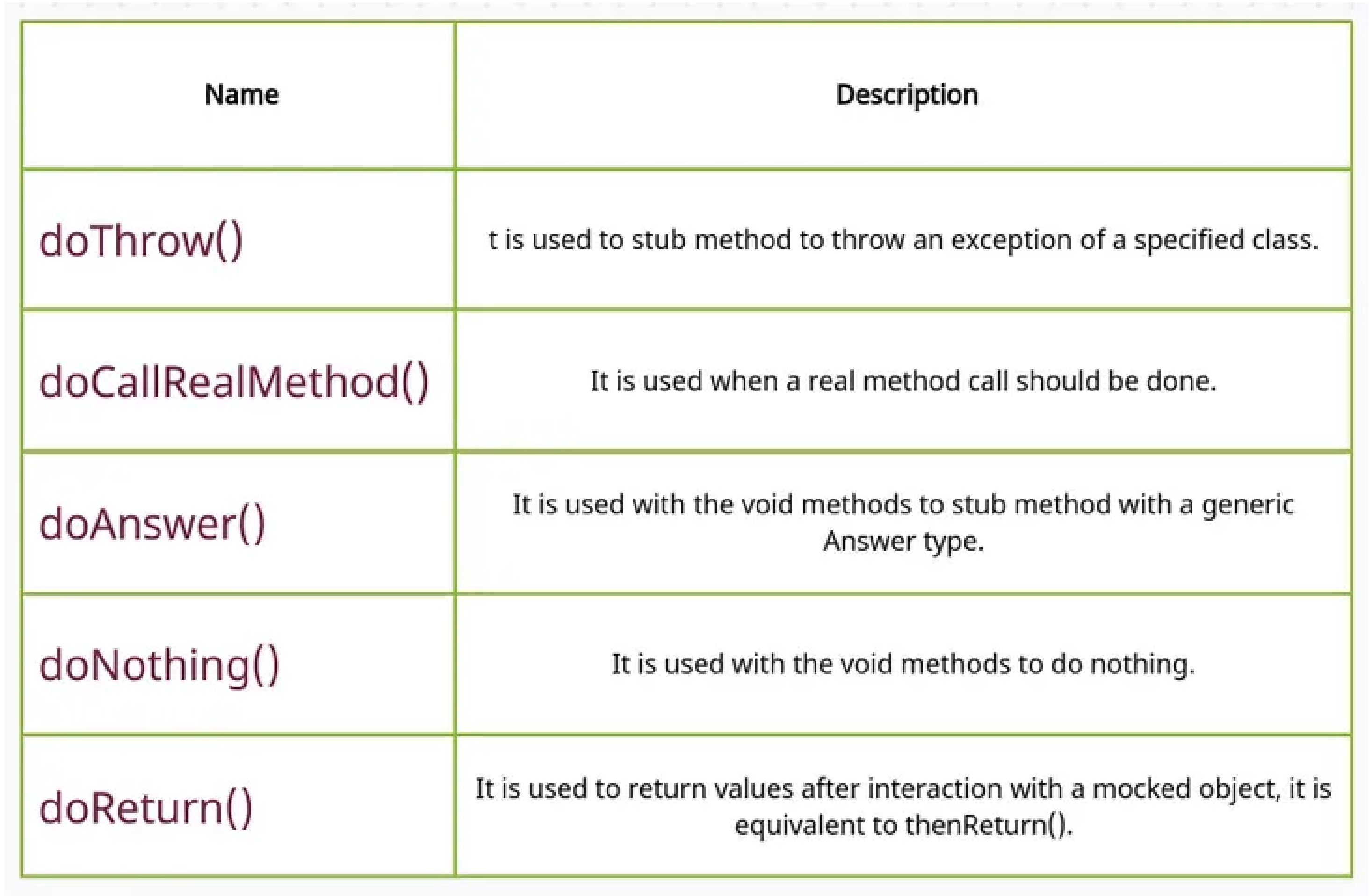
```
Demo demo = mock(Demo.class);
assertEquals(0, demo.getInt());
assertEquals(0,demo.getInteger().intValue()); assertEquals(0d,
demo.getDouble(), 0d); assertFalse(demo.getBoolean());
assertNull(demo.getObject());
assertEquals(Collections.emptyList(),demo.getCollection());
assertNull(demo.getArray()); assertEquals(0L, demo.getStream().count());
assertFalse(demo.getOptional().isPresent());
```

Mockito when()-then method:



Name	Description
thenThrow()	t is used to stub method to throw an exception of a specified class.
thenCallRealMethod()	It is used when a real method call should be done.
doNothing()	It is used with the void methods to do nothing.
thenReturn()	It is used to return values after interaction with a mocked object, it is equivalent to thenReturn().

Different methods used with the do-when approach



Name	Description
doThrow()	t is used to stub method to throw an exception of a specified class.
doCallRealMethod()	It is used when a real method call should be done.
doAnswer()	It is used with the void methods to stub method with a generic Answer type.
doNothing()	It is used with the void methods to do nothing.
doReturn()	It is used to return values after interaction with a mocked object, it is equivalent to thenReturn().

Examples

- Read the following tutorial for more examples
  - Unit tests with Mockito - Tutorial
  - https://www.vogella.com/tutorials/Mockito/article.html

Stubbing consecutive calls

```
@Test
 public void testCalculateBMI() {
    when(bmiService.calculateBMI(1.754,50)).thenReturn("Obese");
    String result1 = bmiCalculator.calculateBMI(1.754,50);
    when(bmiService.calculateBMI(1.754,30)).thenReturn("Under weight");
    String result2 = bmiCalculator.calculateBMI(1.754,30);
    assertEquals("Obese",result1);
    assertEquals("Under weight",result2);
 }
```

for the next real method call, the last stubbing wins

Stub Examples

- `/thenAnswer when(passwordEncoder.encode("1")).thenAnswer( invocation -> invocation.getArgument(0) + "!");`

- `//doAnswer doAnswer(invocation -> invocation.getArgument(0) + "!") .when(passwordEncoder).encode("1");`

References

- https://www.section.io/engineering-education/mocking-with-junit-and-mockito-the-why-and-how/
- https://www.vogella.com/tutorials/Mockito/article.html