[Mockito](https://en.wikipedia.org/wiki/Mockito) is an open source testing framework for Java released under the MIT License. The framework allows the creation of test double objects (mock objects) in automated unit tests for the purpose of test-driven development (TDD) or behavior-driven development (BDD)

Mockito allows developers to verify the behavior of the system under test (SUT) without establishing expectations beforehand. One of the criticisms of mock objects is that there is a tight coupling of the test code to the system under test. Mockito attempts to eliminate the expect-run-verify pattern by removing the specification of expectations. Mockito also provides some annotations for reducing boilerplate code.

**Why Mock?**

All of the code we write has a network of interdependencies, it may call into the methods of several other classes which in turn may call yet other methods; indeed this is the intent and power of object oriented programming. Usually at the same time as writing our feature code we will also write test code in the form of automated unit tests. We use these unit tests to verify the behaviour of our code, to ensure that it behaves as we expect it to behave.

When we unit test our code we want to test it in isolation and we want to test it fast. For the purposes of the unit test we only care about verifying our own code, in the current class under test. Generally we also want to execute our unit tests very regularly, perhaps more than several times per hour when we are refactoring and we are working in our continuous integration environment.

This is when all our interdependencies become an issue. We might end up executing code in another class that has a bug that causes our unit test to fail. Imagine a class which we use to read user details from a database, what happens if there’s no database present when we want to run our unit tests? Imagine a class which calls several remote web services, what if they’re down or take a long time to respond? Our unit tests could fail due to our dependencies and not because of some issue with the behaviour of our code. This is undesirable.

All these issues simply disappear if we use mocks. Mocks act like a substitute for the classes with which we are collaborating, they take their place and behave exactly how we tell them to behave. Mocks let us pretend that our real collaborators are there, even though they aren’t. More importantly mocks can be programmed to return whatever values we want and confirm whatever values are passed to them. Mocks execute instantly and don’t require any external resources. Mocks will return what we tell them to return, throw whatever exceptions we want them to throw and will do these things over and over, on demand. They let us test only the behaviour of our own code, to ensure that our class works, regardless of the behaviour of its collaborators.

There are several mocking frameworks available for Java, each have their own syntax, their own strengths, their own weaknesses. In this tutorial we will be using the Mockito framework, which is one of the more popular mocking frameworks available.

**Collaborators:** In the context of object-oriented programming (OOP) and software design, "class collaborators" refers to other classes or objects that a given class interacts with, uses, or depends on to fulfill its responsibilities.

**Mocking:** Mocking means creating fake versions of objects or classes (of external dependencies) so you can test your code in isolation—without relying on real implementations like databases, APIs, or services

🧪 **Testing in isolation** means evaluating a specific part of a system—like a function, module, or component—separately from the rest, without relying on its external dependencies or interactions. The goal is to make sure that piece works perfectly on its own.

**Stubbing:** This is the process of defining how mock objects should behave. It allows you to specify what values a mock object should return or what exceptions it should throw when particular methods are invoked.

**Verification:** This is used to confirm how mock objects were interacted with, for instance, checking if a method was called a specific number of times or with certain arguments. This ensures that the code under test interacts correctly with its collaborators. It lets us look at the arguments of our mocks to make sure they are as expected. Verification lets us address the other concerns mentioned in the first section – it lets us ensure that exactly the values we expect are passed to our collaborators, and that nothing unexpected happens. Verification lets us determine exactly what happened to the mock.

Mockito does have some limitations, however, including

* You can’t mock final classes
* You can’t mock static methods
* You can’t mock final methods
* You can’t mock equals() or hashCode()

Syntaxes:

// Step 1: Create a mock object for the external API

1. Mockito.mock(): used to create mock objects // static method in Mockito class
   1. Mock object is created for dependency class or interface

YourInterfaceOrClass MockObject = Mockito.mock(YourInterfaceOrClass.class);

// Instantiate the service with the mock dependency

Eg.

mockWeatherApi = Mockito.mock(WeatherApi.class);

weatherService = new WeatherService(mockWeatherApi);

Mock Object can also be created using @Mock annotation (check if reqd)

**Stubbing Syntaxes:**

1. **when().thenReturn():** For methods that return a value.

This is the most common stubbing pattern. It specifies what a mock method should return when invoked.

import static org.mockito.Mockito.when; // Recommended static import

when(mockObject.methodCall(arg1, arg2)).thenReturn(returnValue);

1. **when().thenThrow():** For methods that throw an exception.

Use this to simulate error conditions.

when(mockObject.methodCall()).thenThrow(new SomeException("error message"));

1. **doNothing().when():** For void methods that should do nothing.

By default, void methods on mocks do nothing. You only need to explicitly stub them if you want to change this default behavior (e.g., to throw an exception or call a real method). However, doNothing() is used for clarity or when overriding previous stubbings.

import static org.mockito.Mockito.doNothing;

doNothing().when(mockObject).voidMethodCall(arg1);

**Writing code using Mockito:**

dependency injection (DI) is practically the backbone of mocking, stubbing, and verifying interactions in unit testing.

Here’s why it’s so widely used:

🔄 What Is Dependency Injection (DI)?

* It’s a design pattern where a class receives its dependencies (like services or components) from the outside instead of creating them internally.
* This makes your classes more **modular**, **testable**, and **easy to swap out implementations**.

Yes, what we've been discussing, particularly the use of the constructor to pass in the WeatherApi object to WeatherService, is the most common form of **Dependency Injection (DI)**, specifically **Constructor Injection**.

Let's reiterate why it's called "Dependency Injection":

* **Dependency:** The WeatherService **depends** on a WeatherApi to perform its primary function (getting weather reports). Without a WeatherApi, WeatherService cannot do its job.
* **Injection:** Instead of the WeatherService *creating* its own WeatherApi (which would make it tightly coupled and less flexible and harder to test), the WeatherApi is **"injected"** or **"provided"** to the WeatherService from the *outside*. The WeatherService doesn't know or care *how* the WeatherApi was created; it just receives it.

**How it contrasts with "Traditional" (non-DI) approaches:**

1. **Direct Instantiation (No DI):**

public class WeatherService {

private WeatherApi weatherApi = new RealWeatherApiImpl(); // WeatherService creates its own dependency

public String getWeatherReport(String city) {

return "Current temperature in "+city+" is : "+weatherApi.getTemperature(city);

}

}

Here, RealWeatherApiImpl is a class which implements, weatherApi

**Problem:** Tightly coupled

**Uses of DI:**

**Benefit:** WeatherService no longer creates its dependency. It simply receives it. This makes it:

* **Loosely Coupled:** It depends on the WeatherApi *interface* (abstraction), not a specific RealWeatherApiImpl concrete class.
* **More Testable:** During testing, you can inject a MockWeatherApi instead of a RealWeatherApiImpl.
* **More Flexible:** You can easily swap different WeatherApi implementations without changing WeatherService itself.