Mockito introduction

The aim

Strictly speaking you only want your unit tests to test the exact unit (in Java usually a class) under test, not any external code, and obviously you don't want them to rely on any external dependencies like databases or web access. The advantage of a mocking framework is that it makes replacing dependencies with mocks very easy, so you can concentrate on the questions you want to ask when testing e.g. what is this class's responsibility, what are all the paths through it, what can go wrong, etc. In turn that often makes your code better, as it gives you a chance to see where things could be simpler, or if you're breaking the Single Responsibility Principle, etc. If your code is hard to test, that's a code smell.

Obviously this doesn't mean you can't roll your own mocks, sometimes things are so simple it's easier to do that, but when it's a palaver you can write more productive tests more quickly with a mocking framework.

Using Mockito

There's a good DZone guide to Mockito. Below I've summarised the four main steps, and then I've got a list of basic examples.

Set up a mock

To initiate a mock

```
private InformationProvider mockInformationProvider = Mockito.mock(InformationProvider.class);
```

(There's an annotation way as well, it's a bit more complicated.)

Inject the mock

It's a good design principle to separate instantiation of objects from their use whether or not you're using an actual Dependency Injection framework. (Don't take my word for it, here are Martin Fowler and "Uncle Bob" on the subject.) So supposing your object under test has a constructor which takes its dependencies (dependency injection by constructor) it's as easy as:

```
UsefulClass testObject = new UsefulClass(mockInformationProvider);
```

Specify behaviour of the mock

In the part of the test where you set things up (the bit of the test that gets labelled things like "arrange" in "arrange, act, assert" or "given" in "given, when, then") you can specify how you want your mock to behave when it is called in particular ways. Here's an example:

```
when(mockInformationProvider.evaluate(testBean, “summer”)).thenReturn(25);
```

assuming you've got a variable called testBean. For more see the list.

Verify interations with the mock

Then in the "assert" or "then" part of the test you can verify what has happened. For example, if the bean object should have been stored

```
verify(mockPersistenceMechanism).store(testBean);
```

For more verifications see the list. You may well not need to verify if you've specified behaviour.

Mockito cheatsheet

IntelliJ will help you import the static methods, e.g. Mockito.when()

Behaviour

```
when(mockDependency.calculateValue(testBean)).thenReturn(23);
```

Matching parameters:

```
when(mockDependency.calculateValue(any(Bean.class))).thenReturn(23);
```

You can't mix and match real objects and matchers if the method has more than one argument:

```
when(mockThing.combine(testBean, “autumn”)).thenReturn(otherTestBean);

when(mockThing.combine(any(Bean.class), eq(“autumn”)).thenReturn(otherTestBean);

when(mockThing.combine(eq(testBean), anyString())).thenReturn(otherTestBean);
```

Multiple returns:

```
when(mockDependency.calculateValue(any(Bean.class))).thenReturn(23, 10, 17);
```

returns 23 the first time, 10 the next, and 17 all the times it's called after that.

Throwing exceptions

```
when(mockDependency.calculateValue(testBean)).thenThrow(new IllegalArgumentException());
```

If the method is void, there's a different syntax:

```
doThrow(new RuntimeException()).when(mockThing).consume(testBean);
```

Verification

```
verify(mockThing).consume(testBean);
```

Note that the static method *verify* takes a mock, while the static method *when* takes the return value of an invocation on the mock — i.e. the brackets are different.

How many times:

```
verify(mockThing, never()).consume(testBean);

verify(mockThing, times(3)).consume(any(Bean.class));

verify(mockThing).combine(any(Bean.class), startsWith(“sum”));

verifyZeroInteractions(mockThing, mockDependency);

verifyNoMoreInteractions(mockThing);
```

Capture arguments

Maybe you just want to check that something's true about the parameters of method invocations:

```
ArgumentCaptor<Bean> beanArgCaptor = ArgumentCaptor.forClass(Bean.class);
verify(mockThing, times(3)).consume(beanArgCaptor.capture());
List<Bean> capturedBeans = beanArgCaptor.getAllValues();
capturedBeans.forEach(bean -> assertFalse(bean.isUnwanted());
```

Verify the order of invocations

```
InOrder inOrder = inOrder(mockThing);

inOrder.verify(mockThing).validate(testBean);

inOrder.verify(mockThing).store(testBean);
```

Mockito examples

I have a really basic example I made for some trainees at the Met Office which I have now put up on github. You can browse the repo, or just download the pdf example. It's short and trivial.

I've used mockito in the BetfairTimeformService and BetfairGpsService, which are where you'd expect on svn. In BetfairTimeformService the TimeformApiMeetingsRequestTest is a reasonably straightforward example. In the same repo I had to use a spy in TimeformRequestsManagerTest, which means instead of a mock the test has a real object but can watch what it does and interfere. (Not great unless you have to.) In BetfairGpsService the GpsRaceInformationAnalyzerTest is a long test because I wanted to pin down the complex logic, but again reasonably readable (I hope).

Here's Mockito's webpage, and here is their github repo with readme. They keep their documentation in Javadoc form, but it's more example-based than that implies. There's also a ton of good help on the internet for mockito, and lots on StackOverflow.

Test-Driven Development

If you read up on mocking you'll probably come across the phrase "Don't mock objects you don't own". This comes from Test-Driven Development, TDD, which has the principle that you write tests to make your code design better (the tests are an amiable side effect). In that context, finding yourself mocking something you don't own means that you have a hard dependency on an outside class. You would replace that hard dependency with an interface that specifies what your class needs. (Of course when you write an implementation of that interface using your specific external dependency you're going to need to mock it in the tests, but that class's one responsibility is to handle that external class so it's more acceptable there.)