* JMockit provides us APIs to mock:

1. Public Methods with void and non-void return types
2. Private Methods with void and non-void return types
3. Field Values
4. Static Methods
5. Static Blocks
6. Constructors

* JMockit provides ways to record and verify the mocked values
* JMockit provides a lot of useful and powerful annotations to ease down the testing. For example: @Tested @Injectable, we will be covering few of them in this blog.

Now let us dive into the ocean called JMockit.  
Step 1: Annotations: As we mentioned, there are a number of annotations, so let’s see the most frequently used:

* @**Tested**: This annotation is used to get an initialized object of the class that we want to test along with the injection of the dependencies.
* @**Mocked**: This annotation is used to provide a mock field/parameter for the class being tested. The type of the mock field or parameter can be any kind of **reference type:** an interface, a class (including abstract and final ones), an annotation, or an enum. But **primitive and array types**are not provided (to the class being tested) using @Mocked. A mocked instance of that type is automatically created and assigned to the mock field/parameter.
* **@Injectable**: Indicates that the value of a mock field or mock parameter will be an isolated mocked instance, intended to be passed or *injected* into the code under test. Such instances can be said to be proper *mock objects*, in contrast to the mocked instances of a regular @Mocked type.

So now we got the definitions of both, but in common terms what’s the basic difference between @Mocked and @Injectable?

Expectations recorded inside a new Expectations() {...} block are the regular ones. What this means is that:

* the invocations they specify are expected to occur at least once during the replay phase
* they may occur more than once, though, and in a different order relative to other recorded expectations
* additionally, invocations that don’t match any recorded expectation are allowed to occur in any number and in any order.

In the context of the Faking API, a fake method is any method in a fake class that gets annotated with @Mock. A fake class is any class extending the **mockit.MockUp<T>** generic base class, where T is the type to be faked. The example below shows several fake methods defined in a fake class for our example "real" class,

If no invocation matches a given recorded expectation, a “missing invocation” error gets thrown at the end of the test, causing it to fail (this is only the default behavior, though, as it can be overridden).

The API also supports the concept of strict expectations:

* those that, when recorded, only allow invocations during replay that exactly match the recordings (within explicitly specified allowances, when needed), both in the number of matching invocations (exactly one, by default) and in the order they occur.
* Invocations that occur during replay but fail to match a recorded strict expectation are regarded as unexpected, causing an immediate “unexpected invocation” error, and consequently failing the test.
* **public** **final** **class** FakeLoginContext **extends** MockUp<LoginContext> {
* @Mock
* **public** **void** $init(String name, CallbackHandler callback) {
* assertEquals(**"test"**, name);
* assertNotNull(callback);
* }
* @Mock
* **public** **void** login() {}
* @Mock
* **public** Subject getSubject() { **return** **null**; }
* }
* **public** **final** **class** FakeLoginContext **extends** MockUp<LoginContext> {
* @Mock
* **public** **void** $init(String name, CallbackHandler callback) {
* assertEquals(**"test"**, name);
* assertNotNull(callback);
* }
* @Mock
* **public** **void** login() {}
* @Mock
* **public** Subject getSubject() { **return** **null**; }
* }
* import org.junit.Test;
* import static org.junit.Assert.assertEquals;