# JMockit

An automated testing toolkit for Java

# Introduction

In this tutorial we examine the APIs available in the toolkit, with the help of example tests. The larger API, intended for tests which use mocked dependencies, is the mocking API (also known as the "Expectations" API). Another one is the faking API (aka the "Mockups" API), intended for tests which can benefit from the use of *fake* implementations that avoid the full cost of heavy external components. Finally, there is a small but powerful API that supports out-of-container integration testing.

Even though the tutorial is fairly complete, it does not attempt to cover the entire published APIs in detail. A complete and detailed specification for all public classes, methods, etc. is provided through the API documentation. A copy of this documentation for each version of the toolkit can be found under the "jmockit.github.io/api1x" folder, inside the full distribution zip file. The "`jmockit.jar`" library file (and its Maven equivalent), contains Java source files (with Javadoc comments) for easy access to the API source code and documentation from a Java IDE.

A separate chapter covers the code coverage tool.

## Automated developer testing and test isolation

Software tests written by the developers themselves, to test their own code, are often very important for successful software development. Such tests are usually written with the help of a testing framework, such as JUnit or TestNG; JMockit has specific support for both of these testing frameworks.

Automated developer tests can be divided in two broad categories:

1. **Unit tests**, intended to test a class or component in isolation from the rest of the system.
2. **Integration tests**, intended to test system operations that encompass a unit and its dependencies (other classes/components with which the unit under test interacts).

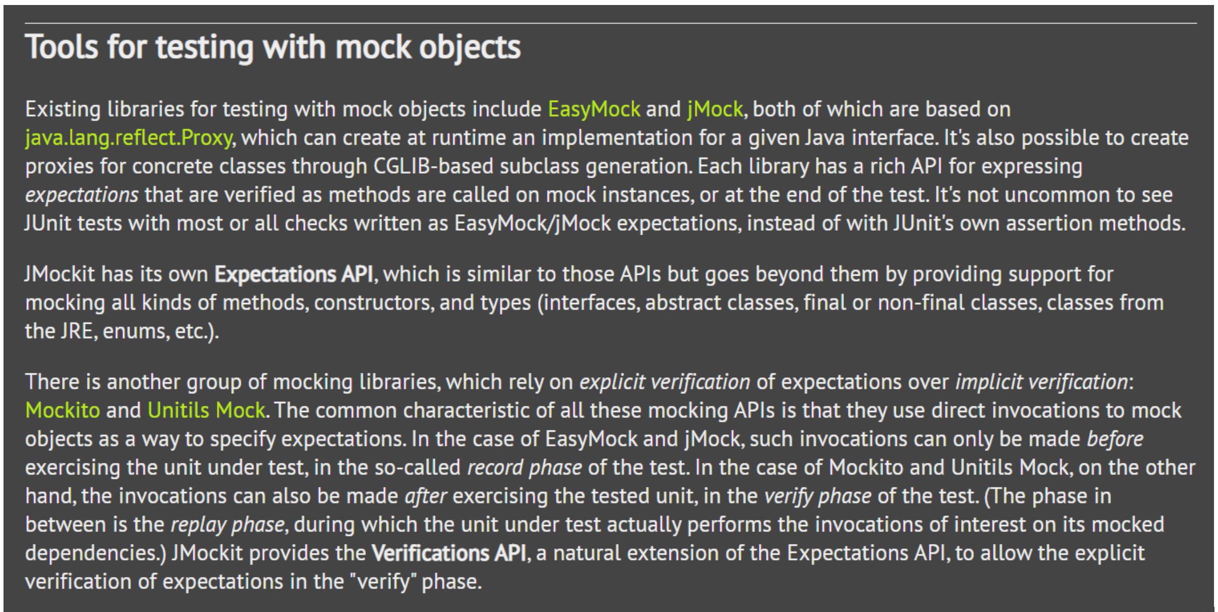
Even though integration tests include the interaction between multiple units, particular tests may not be interested in exercising all components, layers, or sub-systems involved. The ability to isolate the code under test from certain parts of the system is therefore generally useful.

## Testing with mock objects

A common and powerful technique for testing code in isolation is the use of "mocks". Traditionally, a *mock object* is an instance of a class specifically implemented for a single test or set of related tests. This instance is passed to code under test to take the place of one of its dependencies. Each mock object behaves in the way expected by both the code under test and the tests that use it, so that all tests can pass. That, however, is not the only role mock objects usually play. As a complement to the assertions performed by each test, the mock itself can encode additional assertions.

The above is true for conventional mock object tools such as EasyMock, jMock, and Mockito (more about them in the next section). JMockit goes beyond conventional mock objects by allowing methods *and* constructors to be mocked directly on "real" (non-mock) classes, eliminating the need to instantiate mock objects in tests and pass them to code under test; instead, objects created by code under test will execute the mock behavior defined by tests, whenever methods or constructors are called on the real classes. With JMockit, the original implementations of existing methods/constructors are temporarily replaced with mock implementations, usually for the duration of a single test. This mocking approach is completely generic, so that not only `public` instance methods, but also `final` and `static` methods, as well as constructors, can have their implementations replaced, and therefore are also "mockable".

Mocks are most useful for unit tests, but can also be used for integration tests. For example, you may want to test a presentation layer class along with its interactions with other classes in the same layer, without actually depending on the result of calls made to code in other application layers, such as the business or infrastructure layers.

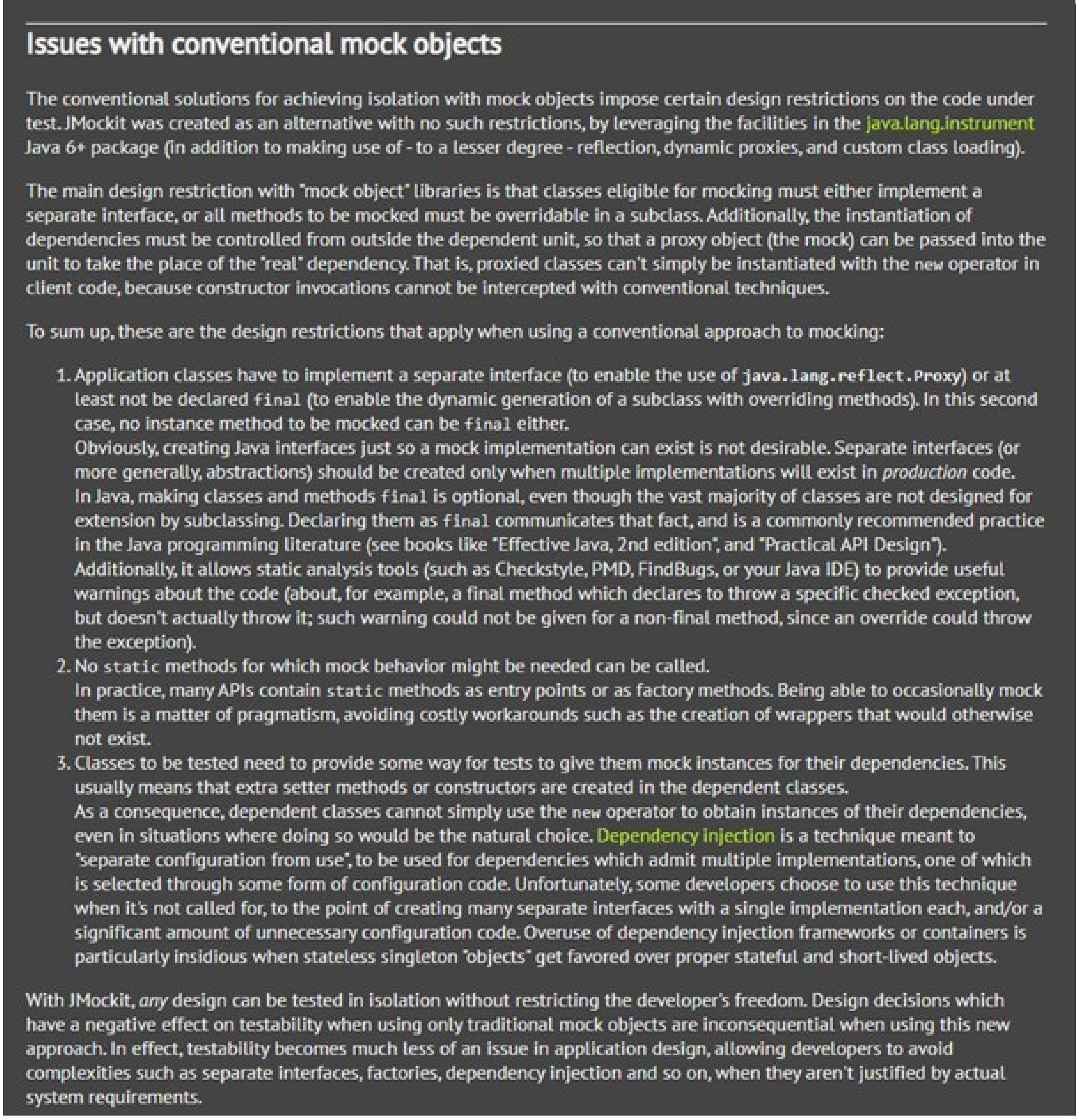


## Tools for testing with mock objects

Existing libraries for testing with mock objects include EasyMock and jMock, both of which are based on java.lang.reflect.Proxy, which can create at runtime an implementation for a given Java interface. It's also possible to create proxies for concrete classes through CGLIB-based subclass generation. Each library has a rich API for expressing *expectations* that are verified as methods are called on mock instances, or at the end of the test. It's not uncommon to see JUnit tests with most or all checks written as EasyMock/jMock expectations, instead of with JUnit's own assertion methods.

JMockit has its own **Expectations API**, which is similar to those APIs but goes beyond them by providing support for mocking all kinds of methods, constructors, and types (interfaces, abstract classes, final or non-final classes, classes from the JRE, enums, etc.).

There is another group of mocking libraries, which rely on *explicit verification* of expectations over *implicit verification*: Mockito and Unitils Mock. The common characteristic of all these mocking APIs is that they use direct invocations to mock objects as a way to specify expectations. In the case of EasyMock and jMock, such invocations can only be made *before* exercising the unit under test, in the so-called *record phase* of the test. In the case of Mockito and Unitils Mock, on the other hand, the invocations can also be made *after* exercising the tested unit, in the *verify phase* of the test. (The phase in between is the *replay phase*, during which the unit under test actually performs the invocations of interest on its mocked dependencies.) JMockit provides the **Verifications API**, a natural extension of the Expectations API, to allow the explicit verification of expectations in the "verify" phase.



## Issues with conventional mock objects

The conventional solutions for achieving isolation with mock objects impose certain design restrictions on the code under test. JMockit was created as an alternative with no such restrictions, by leveraging the facilities in the java.lang.instrument Java 6+ package (in addition to making use of - to a lesser degree - reflection, dynamic proxies, and custom class loading).

The main design restriction with "mock object" libraries is that classes eligible for mocking must either implement a separate interface, or all methods to be mocked must be overridable in a subclass. Additionally, the instantiation of dependencies must be controlled from outside the dependent unit, so that a proxy object (the mock) can be passed into the unit to take the place of the "real" dependency. That is, proxied classes can't simply be instantiated with the `new` operator in client code, because constructor invocations cannot be intercepted with conventional techniques.

To sum up, these are the design restrictions that apply when using a conventional approach to mocking:

1. Application classes have to implement a separate interface (to enable the use of `java.lang.reflect.Proxy`) or at least not be declared `final` (to enable the dynamic generation of a subclass with overriding methods). In this second case, no instance method to be mocked can be `final` either.
   Obviously, creating Java interfaces just so a mock implementation can exist is not desirable. Separate interfaces (or more generally, abstractions) should be created only when multiple implementations will exist in *production* code.
   In Java, making classes and methods `final` is optional, even though the vast majority of classes are not designed for extension by subclassing. Declaring them as `final` communicates that fact, and is a commonly recommended practice in the Java programming literature (see books like "Effective Java, 2nd edition", and "Practical API Design").
   Additionally, it allows static analysis tools (such as Checkstyle, PMD, FindBugs, or your Java IDE) to provide useful warnings about the code (about, for example, a final method which declares to throw a specific checked exception, but doesn't actually throw it; such warning could not be given for a non-final method, since an override could throw the exception).
2. No `static` methods for which mock behavior might be needed can be called.
   In practice, many APIs contain `static` methods as entry points or as factory methods. Being able to occasionally mock them is a matter of pragmatism, avoiding costly workarounds such as the creation of wrappers that would otherwise not exist.
3. Classes to be tested need to provide some way for tests to give them mock instances for their dependencies. This usually means that extra setter methods or constructors are created in the dependent classes.
   As a consequence, dependent classes cannot simply use the `new` operator to obtain instances of their dependencies, even in situations where doing so would be the natural choice. Dependency injection is a technique meant to "separate configuration from use", to be used for dependencies which admit multiple implementations, one of which is selected through some form of configuration code. Unfortunately, some developers choose to use this technique when it's not called for, to the point of creating many separate interfaces with a single implementation each, and/or a significant amount of unnecessary configuration code. Overuse of dependency injection frameworks or containers is particularly insidious when stateless singleton "objects" get favored over proper stateful and short-lived objects.

With JMockit, *any* design can be tested in isolation without restricting the developer's freedom. Design decisions which have a negative effect on testability when using only traditional mock objects are inconsequential when using this new approach. In effect, testability becomes much less of an issue in application design, allowing developers to avoid complexities such as separate interfaces, factories, dependency injection and so on, when they aren't justified by actual system requirements.

## An example

Consider a business service class which provides a business operation with the following steps:

1. find certain persistent entities needed by the operation
2. persist the state of a new entity
3. send a notification e-mail to an interested party

The first two steps require access to the application database, which is done through a simplified API to the persistence subsystem. The third one can be achieved with a third-party API for sending e-mail, which in this example is Apache's Commons Email library.

Therefore, the service class has two separate dependencies, one for persistence and another for e-mail. In order to unit test the business operation while verifying proper interaction with these collaborators, we will use the mocking API. Complete source code for a working solution - with all tests - is available online.

```java
package jmockit.tutorial.domain;

import java.math.*;
import java.util.*;
import org.apache.commons.mail.*;
import static jmockit.tutorial.persistence.Database.*;

public final class MyBusinessService
{
   private final EntityX data;

   public MyBusinessService(EntityX data) { this.data = data; }

   public void doBusinessOperationXyz() throws EmailException
   {
      List<EntityX> items =
(1)      find("select item from EntityX item where item.someProperty = ?1", data.getSomeProperty());

      // Compute or obtain from another service a total value for the new persistent entity:
      BigDecimal total = ...
      data.setTotal(total);

(2)   persist(data);

      sendNotificationEmail(items);
   }

   private void sendNotificationEmail(List<EntityX> items) throws EmailException
   {
      Email email = new SimpleEmail();
      email.setSubject("Notification about processing of ...");
(3)   email.addTo(data.getCustomerEmail());

      // Other e-mail parameters, such as the host name of the mail server, have defaults defined
      // through external configuration.

      String message = buildNotificationMessage(items);
      email.setMsg(message);

(4)   email.send();
   }

   private String buildNotificationMessage(List<EntityX> items) { ... }
}
```

The `Database` class contains only static methods and a private constructor; the `find` and `persist` methods should be obvious, so we won't list them here (assume they are implemented on top of an ORM API, such as JPA).

So, how can we unit test the "doBusinessOperationXyz" method without making any changes to the existing application code? In the following JUnit test class, each test will verify the invocations of interest made from the unit under test to its external dependencies. These invocations are the ones at points `(1)`-`(4)` indicated above.

```
package jmockit.tutorial.domain;

import org.apache.commons.mail.*;
import jmockit.tutorial.persistence.*;

import org.junit.*;
import mockit.*;

public final class MyBusinessServiceTest
{
   @Mocked(stubOutClassInitialization = true) final Database unused = null;
   @Mocked SimpleEmail anyEmail;

   @Test
   public void doBusinessOperationXyz() throws Exception
   {
      final EntityX data = new EntityX(5, "abc", "abc@xpta.net");
      final EntityX existingItem = new EntityX(1, "AX5", "someone@somewhere.com");

      new Expectations() {{
(1)      Database.find(withSubstring("select"), any);
         result = existingItem; // automatically wrapped in a list of one item
      }};

      new MyBusinessService(data).doBusinessOperationXyz();

(2)   new Verifications() {{ Database.persist(data); }};
(4)   new Verifications() {{ anyEmail.send(); times = 1; }};
   }

   @Test(expected = EmailException.class)
   public void doBusinessOperationXyzWithInvalidEmailAddress() throws Exception
   {
      new Expectations() {{
(3)      anyEmail.addTo((String) withNotNull()); result = new EmailException();
      }};

      EntityX data = new EntityX(5, "abc", "someone@somewhere.com");
      new MyBusinessService(data).doBusinessOperationXyz();
   }
}
```

In a behavior-oriented mocking API like this, each test can be divided in three consecutive execution steps: *record*, *replay*, and *verify*. Recordings occur from inside an *expectation recording block*, while verifications occur from inside an *expectation verification block*; replays are invocations from inside the code under test. Note, however, that only invocations to *mocked* methods count; methods (or constructors) belonging to types or instances with no associated mock field or mock parameter are not mocked, and therefore cannot be recorded nor verified.

As the example test above shows, recording and verifying expectations is achieved simply by invoking the desired methods (as well as constructors, even if not shown here) from inside a recording or verification block. Matching of method arguments can be specified through API fields such as "`any`" and "`anyString`", or through API methods such as "`withNotNull()`". Values to return (or exceptions to throw) from matching invocations during replay are specified during recording through the "`result`" field. Invocation count constraints can be specified, either when recording or when verifying, through API field assignments like "`times = 1`".

## Running tests with JMockit

To run tests that use any of the JMockit APIs, use your Java IDE, Ant/Maven script, etc. the way you normally would. In principle, any JDK of version **1.6** or newer, on Windows, Mac OS X, or Linux, can be used. JMockit supports (and requires) the use of JUnit or TestNG; details specific to each of these test frameworks are as follows:

- For **JUnit 4.5+** test suites, make sure that `jmockit.jar` appears *before* the JUnit jar in the classpath. Alternatively, annotate test classes with `@RunWith(JMockit.class)`.
  (Note for **Eclipse** users: when specifying the order of jars in the classpath, make sure to use the "Order and Export" tab of the "Java Build Path" window. Also, make sure the Eclipse project uses the JRE from a JDK installation instead of a "plain" JRE, since the latter lacks the "`attach`" native library.)
- For **TestNG 6.2+** and **JUnit 5+** test suites, simply add `jmockit.jar` to the classpath (at any position).

In other situations (like running on JDK implementations other than the Oracle JDKs), you may need to pass "`-javaagent:`*`<proper path>`*`/jmockit.jar`" as a JVM initialization parameter. This can be done in the "Run/Debug Configuration" for both Eclipse and IntelliJ IDEA, or with build tools such as Ant and Maven.

### Running tests with the JUnit Ant task

When using the `<junit>` element in a `build.xml` script, it's important to use a separate JVM instance. For example, something like the following:

```
<junit fork="yes" forkmode="once" dir="directoryContainingJars">
   <classpath path="jmockit.jar"/>

   <!-- Additional classpath entries, including the appropriate junit.jar -->

   <batchtest>
      <!-- filesets specifying the desired test classes -->
   </batchtest>
</junit>
```

### Running tests from Maven

The JMockit artifacts are located in the central Maven repository. To use them in a test suite, add the following to your `pom.xml` file:

```
<dependencies>
   <dependency>
      <groupId>org.jmockit</groupId>
      <artifactId>jmockit</artifactId>
      <version>1.x</version>
      <scope>test</scope>
   </dependency>
</dependencies>
```

Make sure the specified version is the one you actually want. Find the current version in the development history page. When using JUnit 4, this dependency should come *before* the "`junit`" dependency.

For information on using JMockit Coverage with Maven, see the relevant section in that chapter.