First steps to Lmock

Summary

In this tutorial, we assume that you are already familiar with mocking techniques. If not, you can get many interesting information by reading:

- The wikipedia page describing mocks: http://en.wikipedia.org/wiki/Mock_object
- Martin Fowler's article about mocks: http://martinfowler.com/articles/mocksArentStubs.html

Since Lmock gets its inspiration from the most popular mocking frameworks, you may also be interested in looking at Jmock (http://www.jmock.org), EasyMock (http://easymock.org/) and Mockito (http://code.google.com/p/mockito/).

Caution: In this tutorial, the examples are written with JUnit4. Lmock is fully compatible with Android, thus JUnit3 too.

The basic concepts of Lmock

Lmock relies on the following paradigm to design your tests: “a test tells a **story** that must comply with a **scenario** defined as a series of **expectations** using **mocks**”.

In other words, a test requires:

- Mocks for the different interfaces you need to fake
- A scenario, by creating expectations
- The test contents, representing the story

If ever the story is incoherent with the scenario, Lmock raises an exception at runtime.

The mocks are created by the **Mock factory**, as illustrated below:

```
1. import com.vmware.lmock.impl.Mock;
2. public class SimpleMockExample {
3.       @SuppressWarnings("unchecked")
4.       private final List<String> stringList = Mock.getObject(List.class);
5.       ...
6. }
```

This example creates a mock called `stringList`, faking an object implementing the standard Java interface `List`.

Remember that mocks always apply on interfaces, not classes. If you try to mock a class, you'll end up with a runtime exception!

The next example shows a trivial case implying scenarios and stories:

```
1. import com.vmware.lmock.impl.Mock;
2. import com.vmware.lmock.impl.Scenario;
3. import com.vmware.lmock.impl.Story;
4. import static com.vmware.lmock.checker.Occurrences.*;
5.
6. public class SimpleMockExample {
7.     @SuppressWarnings("unchecked")
8.     private final List<String> stringList = Mock.getObject(List.class);
9.
10.          Scenario scenario = new Scenario() {
11.              {
12.                  expect(stringList).get(0);
13.                  willReturn(null).occurs(exactly(1));
14.                  expect(stringList).add("hello");
15.                  occurs(exactly(1));
16.              }
17.          };
18.
19.          @Test
20.          public void testWillPass() {
21.              Story story = Story.create(scenario);
22.              story.begin();
23.              if (stringList.get(0) == null) {
24.                  stringList.add("hello");
25.              }
26.              story.end();
27.          }
28.
29.          @Test
30.          public void testWillFail() {
31.              Story story = Story.create(scenario);
32.              story.begin();
33.              stringList.add("hello");
34.              story.end();
35.          }
36.      }
37.
```

The line 10 defines a scenario that will be shared by several tests:

- Line 12, 13: we expect that the user tries to get the very first element of the list (`expect(stringList).get(0)`)
  - as a result, he will get a null element (`willReturn(null)`)
  - this operation must occur exactly 1 time (`occurs(exactly(1))`).
- Line 14, 15: we expect that the user adds a string into the list. This operation must occur exactly one time.

The first test (`testWillPass`) runs a story complying with the scenario.

This story is:

- Created and associated to the scenario, using the factory method `Story.create`
- Surrounded by a start point (`story.begin()`) and an end-point (`story.end()`)

Naturally, stories may begin but never end, for example because an exception occurs or an assertion fails. In fact, the goal of the `end` method is to verify that when the story finishes, the scenario is actually completed (this is the equivalent of `assertIsSatisfied` in Jmock).

The second test (`testWillFail`) fails, because the user does not call `stringList.get(0)`, while it was expected to occur exactly 1 time. As a result, a runtime exception is raised, providing information about the progress of Lmock regarding the scenario:

```
com.vmware.lmock.exception.UnsatisfiedOccurrenceError: expectation error:
expectation 'interface java.util.List.get(Integer=0):returns(null)/[1..1]' was not fully satisfied
what happened up to now:
satisfied 0 times: interface java.util.List.get(Integer=0):returns(null)/[1..1]
...
```

*Note: you may feel a bit uncomfortable with the above syntax, requesting to explicitly create a scenario and assign a story... In fact, Lmock provides a much concise way to define your scenarios and stories, by creating **masquerades**. Patience, we'll soon explain how to create your masquerades.*

Stories behind the scenes

To properly design your tests you may think of scenarios as checklists of expectations and a story as a **pointer** progressing through this checklist. Whenever your test is invoking the method of a mock,

Lmock checks if this **invocation satisfies the pointed expectation**. If not, an exception is raised, otherwise, a specific result is returned and the pointer progresses.

Let's consider, for example, the scenario:

	Invocation	Must happen...	Result
1	`StringList.get(0)`	Exactly 1 time	`null`
2	`stringList.add("hello");`	Exactly 1 time	Not applicable (`void`)
3	`stringList.get(0);`	Exactly 1 time	`"hello"`

The corresponding code is:

```
Story story = Story.create(new Scenario() {
    {
        expect(stringList).get(0);
        occurs(1);
        willReturn(null);
        expect(stringList).add("hello");
        occurs(1);
        expect(stringList).get(0);
        willReturn("hello");
        occurs(1);
    }
});
```

Let's run the following test:

```
1.          story.begin();
2.          if (stringList.get(0) == null) {
3.              stringList.add("hello");
4.              assertEquals("hello", stringList.get(0));
5.          }
6.          story.end();
```

On line 2, the test satisfies the first expectation of the checklist. Since the invocation must happen exactly one time, the pointer is moved to the second expectation of the checklist.

One line 3, the test satisfies the second expectation of the checklist. Since the invocation must happen exactly one time, the pointer is moved to the third expectation of the checklist.

On line 4, the test satisfies the third expectation of the checklist. Since the invocation must happen exactly one time, the pointer is moved to the end of the checklist. No additional invocation should happen now.

Now, let's run the test:

```
1.          story.begin();
2.          if (stringList.get(0) == null) {
3.              stringList.add("hello world");
4.              assertEquals("hello", stringList.get(0));
5.          } else {
6.              fail("was not expecting data in the list...");
7.          }
8.          story.end();
```

On line 2, the test satisfies the first expectation of the checklist. Since the invocation must happen exactly one time, the pointer is moved to the second expectation of the checklist.

One line 3, the test performs an invocation that doesn't match the expectation currently pointed at (the argument should be "`hello`", not "`hello world`"). Lmock considers that it cannot move the pointer, since the expected invocation must happen exactly one time.

As a consequence, it throws an exception:

```
com.vmware.lmock.exception.UnsatisfiedOccurrenceError: expectation error: expectation 'interface
java.util.List.add(String=hello):void/[1..1]' was not fully satisfied

what happened up to now:

satisfied 1 times: interface java.util.List.get(Integer=0):returns(null)/[1..1]

satisfied 0 times: interface java.util.List.add(String=hello):void/[1..1]

       etc..
```

You may notice that the exception comes with a summary of the checklist ("`what happened up to now`"). This is called the **story track**. You may read that story track at any moment by calling the static method `StoryTrack.get()`.

Clauses

In the Lmock terminology, a **clause** represents any method contributing to the specification of an expectation (or a **stub**, as explained later in this tutorial).

Up to now, you have discovered a certain number of such clauses:

- `expect`: creates a new expectation on a given mock
- `occurs`: specifies how an expectation must occur to let the pointer progress
- `willReturn`: defines the value returned by the mocked invocation

More formally, Lmock defines clauses to

- Manage expectations
- Define the prototypes of the expected methods
- Specify how expectations must occur
- Specify the results of invocations.

Except for the second case, all the clauses return the expectation under construction and so can be included in cascaded invocations.

In other words, you can indifferently write:

```
expect(stringList).get(0);
occurs(1);
willReturn(null);
```

Or:

```
expect(stringList).get(0);
expect().occurs(1).willReturn(null);
```

Or even:

```
expect(stringList).get(0);
occurs(1).willReturn(null);
```

That's only a matter of taste...

Managing expectations with clauses

The `expect` clause comes in two forms:

- To create a new expectation for a given mock, use `expect(mock)` and append the invoked method
- To remind the expectation under construction, use `expect()`, like in "`expect().occurs(1)`".

Defining prototypes of invocations with clauses

In all our previous examples, the invoked methods were expressed with specific argument values. This is somehow too restrictive.

For that reason, Lmock defines the **argument clauses** (sometimes called *with clauses*), which allow to define more generic types of arguments:

- `with(value)` specifies that *value* is expected as argument
- `anyOf(type)` specifies that any object of *type* (or an inheriting class) is expected
- `aNonNullOf(type)` is the same as `anyOf(type)`, except that the argument must not be null
- `with(checker)` allows to defer the checking of arguments to specific mechanism, called `Checker`, described later in this tutorial.

It is very important to remember that **you cannot mix argument clauses with specific argument values**.

For example, the expression

```
expect(stringList).add(1, anyOf(String.class));
```

Is wrong... It must be specified as:

```
expect(stringList).add(with(1), anyOf(String.class));
```

Defining the occurrences of an expectation

The `occurs` clause defines how many times the expectation must happen in the scenario. The pointer will not be able to move through the checklist until the expressed condition is verified.

This clause takes an `Occurrences` object as input, assuming that such an object is provided by one of the factory method described here-in (you may, in fact, statically import them).

By default, Lmock supposes that an expectation can happen any number of times (including never). This is represented by `Occurrences.any()`.

You may explicitly specify that an invocation is not expected to occur, using the clause `Occurrences.never()`. This is only a convenience syntactic features, since it's equivalent to not specifying an expectation for that invocation at all.

Other types of occurrences define a boundary in the number of allowed invocations:

- `Occurrences.exactly(N)` : must be called exactly *N* times
- `Occurrences.atLeast(N)` : must be called at least *N* times
- `Occurrences.atMost(N)` : must not be called more than *N* times
- `Occurrences.between(N,M)` : must be called at least *N* times and not more than *M* times

Note: because the `exactly` clause is the most frequent one in tests, Lmock defines `occurs(N)` as an equivalent to `occurs(exactly(N))`.

Defining your own occurrence checker

Under specific circumstances, you may feel that the standard occurrence schemes do not fit your purpose. In this case, you may want to create your own controller and provide it to the `occurs` clause.

To do so, you must create your own instance of `OccurrenceChecker`, implementing:

- `hasReachedLimit`: no more invocation is allowed, the pointer cannot remain on the current expectation
- `canEndNow`: there were enough invocations to satisfy the expectation, the pointer can (but is not required to) move to the next expectation

Defining the result of an invocation

As already mentioned, you can define the value returned by the invocation of a mock using the `willReturn` clause. You may also throw an exception when such an invocation occurs, thanks to the `willThrow` clause.

For example:

```
expect(stringList).get(1);
willThrow(new IndexOutOfBoundsException());
```

In fact those two classes are a shortcut to the more generic clause `will`:

- `willReturn(`*`value`*`)` is equivalent to `will(returnValue(`*`value`*`))`
- `willThrow(`*`exception`*`)` is equivalent to `will(throwException(`*`exception`*`))`

The `returnValue` and `throwException` methods are provided by `com.vmware.lmock.impl.InvocationResult`.

You may finally delegate the generation of such a result to a specific method. To do this you need to create an `InvocationResultProvider` and use `willDelegateTo`.

Be very careful when doing this, in particular with the fact that:

- You should not invoke any mock in such a procedure, otherwise Lmock behavior can be surprising
- You must return a value or throw an exception compatible with the invocation that has been trapped

Stubs

Mocks are an advantageous alternative to the traditional Java stubbing model. Basically, this model consists in defining fake implementations of classes or interfaces. The major drawback is that when you want to stub very few functions in the whole interface, you must implement all the expected methods of that interface.

Lmock offers an alternative by **defining stubs only for the methods you actually need in your test**.

For example, let's consider an interface providing a global context of internationalization:

```
interface IntlContext {
    String getLanguage();
    String getCountry();
    String getCurrency();
    String getTimezone();
    String getDateTimeFormat();
    String getMetricSystem();
    String getTemperatureUnit();
}
```

Imagine that you want to test an object (class `TimeUtil`) that picks the current time and displays it with the proper format on the proper timezone via the method `displayTime`.

```
class TimeUtil {
    private final IntlContext context;

    TimeUtil(IntlContext context) {
        this.context = context;
    }

    void displayTime() {
        System.out.println("using intl context: " + context.getTimezone() +
              " " + context.getDateTimeFormat());
        // Implement the actual method, using the context.
    }
}
```

We assume that `displayTime` only uses `getTimezone` and `getDateTimeFormat`.

Of course we could envision to create expectations simulating the invocation of those two methods. But in some cases, this may be off-putting and difficult to maintain. This is where stubs are useful: instead of rewriting the expectation each time, you can create a **single** stub that will be called whenever the invocation occurs, no matter when it occurs:

```
1.      import com.vmware.lmock.impl.Stubs;
2.      …
3.      @Test
4.      void testDisplayTime() {
5.          final IntlContext context = Mock.getObject(IntlContext.class);
6.          Stubs stubs = new Stubs() {
7.              {
8.                  stub(context).getDateTimeFormat();
9.                  willReturn("yy/mm/dd");
10.                 stub(context).getTimezone();
11.                 willReturn("UTC+1");
12.             }
13.             };
14.
15.         Story story = Story.create(new Scenario(), stubs);
16.         story.begin();
17.         TimeUtil tu = new TimeUtil(context);
18.         tu.displayTime();
19.         story.end();
20.     }
```

From line 6 to 13, you can see that stubs are created the same way than expectations are, except that the `stub` clause is used to define a new stub.

On line 15, we create a new story... In the context of this example, we do not need a scenario (the first argument is an empty scenario, it could also be `null`) and add the stubs to the story.

Stories behind the scene: how stubs are managed

The overall policy regarding the management of stubs and expectations is the following: assuming the invocation of a mock during a story, Lmock:

- First searches for a stub that could be satisfied by this invocation
- If no stub is found, check if the pointed expectation is satisfied by the invocation
- Throw an exception if none of the above condition is satisfied

You may observe, in particular when using argument clauses, that several stubs may satisfy the same invocation. In order to remove this ambiguity, Lmock always puts the priority to the most recently declared stub.

Next is an illustration of this priority scheme:

```
1. public class PriorityExample {
2.     @Test
3.     public void testStubAndExpectationPriority() {
4.         @SuppressWarnings("unchecked")
5.         final List<String> stringList = Mock.getObject(List.class);
6.         Story story = Story.create(new Scenario() {
7.             {
8.                 expect(stringList).indexOf(anyOf(String.class));
9.                 willReturn(0);
10.                occurs(1);
11.            }}, new Stubs() {
12.                  {
13.                stub(stringList).indexOf(aNonNullOf(String.class));
14.                willReturn(1);
15.                stub(stringList).indexOf("hello world");
16.                willReturn(2);
17.            }});
18.
19.            story.begin();
20.            assertEquals(2, stringList.indexOf("hello world"));
21.            assertEquals(1, stringList.indexOf("hello again!"));
22.            assertEquals(0, stringList.indexOf(null));
23.            story.end();
24.      }
25.      }
```

On lines 8, 9, 10, the scenario expects that there will be one invocation of `stringList.indexOf`, with any string as input. Because stubs prevail on expectations, such an invocation is checked if and only if no stub refers to `stringList.indexOf`.

There are in fact two such stubs on line 13 and 15. Because the latest stub prevails on the previous ones, the final rule behavior will be:

- If `stringList.indexOf` is invoked with the argument "`hello world`", apply the stub declared on line 15
- Else if the argument to `stringList.indexOf` is non null, apply the stub declared on line 13
- Finally, use the expectation declared on line 8 (which means, if the argument is null)

Re-using scenarios and stubs

Lmock defines several mechanisms that allow you to re-use the same scenario or stubs in different stories (thus saving the effort of re-writing things each time).

The most obvious one is `Story.create`, since it takes one scenario and any number of stubs as arguments (so you can create several stories with the same scenario and the same combination of stubs).

An alternative is to use the `append` clause when constructing your scenario or stubs.

In the next example, the test includes a predefined set of expectations using this clause:

```
1. import static com.vmware.lmock.checker.Occurrences.atLeast;
2. import com.vmware.lmock.impl.Mock;
3. import com.vmware.lmock.impl.Scenario;
4. import com.vmware.lmock.impl.Story;
5. ...
6.
7. public class AppendExample {
8.     @SuppressWarnings("unchecked")
9.     private final List<String> list = Mock.getObject(List.class);
10.          private final Scenario expectFirstElementReturnsNull =
11.             new Scenario() {
12.              {
13.                  expect(list).get(0);
14.                  expect().willReturn(null).occurs(atLeast(1));
15.              }
16.          };
17.
18.          @Test
19.          public void testWithAppend() {
20.              Story story = Story.create(new Scenario() {
21.                  {
22.                      append(expectFirstElementReturnsNull);
23.                      expect(list).add("hello!");
24.                      occurs(1);
25.                  }
26.              });
27.
28.              story.begin();
29.              assertNull(list.get(0));
30.              list.add("hello!");
31.              story.end();
32.          }
33.      }
```

Default invocation handlers

If we strictly follow the logic of mocking and stubbing, then **every method of a mock should be stubbed or expected**.

In practice, this sometimes generates more annoyance than benefits, in particular when considering the three common methods `toString`, `equals` and `hashCode`... The solution brought by Lmock is to provide a default implementation for those three functions, based on the underlying mock objects:

- `toString` represents the mock as "`Mock(`*`class`*`)$`*`id`*", where *`class`* is the class of the faked interface and *`id`* is a unique identifier of the mock
- `equals` and `hashCode` are based on the equality of mocks

This said, some tests may need to implement their own versions of those functions. Lmock allows to overwrite the default handlers by explicitly defining a stub or an expectation for those methods, as illustrated here-after:

```
1. public class DefaultInvocationHandlersExample {
2.     @SuppressWarnings("unchecked")
3.     final List<String> list1 = Mock.getObject(List.class);
4.     @SuppressWarnings("unchecked")
5.     final List<String> list2 = Mock.getObject(List.class);
6.
7.     @Test
8.     public void testDefaultInvocationHandlerExample() {
9.         // The default implementation of common methods:
10.             assertFalse(list1.equals(list2));
11.             assertEquals("Mock(List)$0", list1.toString());
12.             assertEquals("Mock(List)$1", list2.toString());
13.
14.             // Overwrite those methods and replay the test:
15.             Story story = Story.create(new Scenario(), new Stubs() {
16.                 {
17.                     stub(list1).equals(list2);
18.                     willReturn(true);
19.                     stub(list1).toString();
20.                     willReturn("list#1");
21.                     stub(list2).toString();
22.                     willReturn("list#2");
23.                 }
24.             });
25.
26.             story.begin();
27.             assertEquals(list1, list2);
28.             assertEquals("list#1", list1.toString());
29.             assertEquals("list#2", list2.toString());
30.             story.end();
31.         }
32.     }
```

Checkers

Under many circumstances, the default argument clauses provided by Lmock are not sufficient to write your expectations.

For example, let's consider an entity representing users in a database:

```
class User {
    private final String firstName, lastName;

    User(String firstName, String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }

    String lastName() {
        return lastName;
    }
```

```
        String firstName() {
            return firstName;
        }
    }
```

And assume a DAO layer that persists users:

```
    interface UserDao {
        void persist(User user);
    }
```

We would like to test a service layer able to create and persist users, based on the DAO:

```
    class UserDaoService {
        private final UserDao dao;

        UserDaoService(UserDao dao) {
            this.dao = dao;
        }

        void createUser(String firstName, String lastName) {
            dao.persist(new User(firstName, lastName));
        }
    }
```

One obvious test would be to call create user and verify that it actually persists a new user entity (with the same first and last name).

To do so, we will create a mock for the DAO layer and specify an expectation on a specific user entity. The problem is to check that entity argument. We could override the equals method of the `User` entity, but that would mean an “intrusion” in the tested program, which is not acceptable (a test remains a test).

The solution is to create a checker for the `User` entity, validating the expected first name and last name:

```
import com.vmware.lmock.checker.Checker;

…
    class UserChecker implements Checker<User> {
        private final String firstName, lastName;

        UserChecker(String firstName, String lastName) {
            this.firstName = firstName;
            this.lastName = lastName;
        }

        @Override
        public boolean valueIsCompatibleWith(User value) {
            return firstName.equals(value.firstName())
                    && lastName.equals(value.lastName());
        }

        @Override
        public Class<?> getRelatedClass() {
            return User.class;
        }
    }
```

Thanks to this checker, validating the invocations to the persist method becomes quite easy: simply use an argument clause, with a user checker as argument:

```
    @Test
    public void testUserChecker() {
        final UserDao dao = Mock.getObject(UserDao.class);
        Story story = Story.create(new Scenario() {
            {
                expect(dao).persist(with(new UserChecker("john", "doe")));
                occurs(1);
            }
        });

        story.begin();
        UserDaoService instance = new UserDaoService(dao);
        instance.createUser("john", "doe");
        story.end();
    }
```

Note: before writing a new checker, you are encouraged to see if one of the default checkers provided by the package `com.vmware.checker` could fit you requirements.

Get rid of the ceremonials, create masquerades

The goal of **masquerades** is to provide a more straightforward method to write scenarios, stubs and stories, by mixing all the concepts in your code.

Up to now, we have seen that this required to follow a certain paradigm: create the scenario, create the stubs, create the story, begin the story, end the story. A masquerade hides all the implementation details

by offering a much simpler syntax.

A masquerade is driven by a **schemer** in charge of automatically managing a story and inject stubs and expectations on the basis of your **directives**.

In other words, rather than writing:

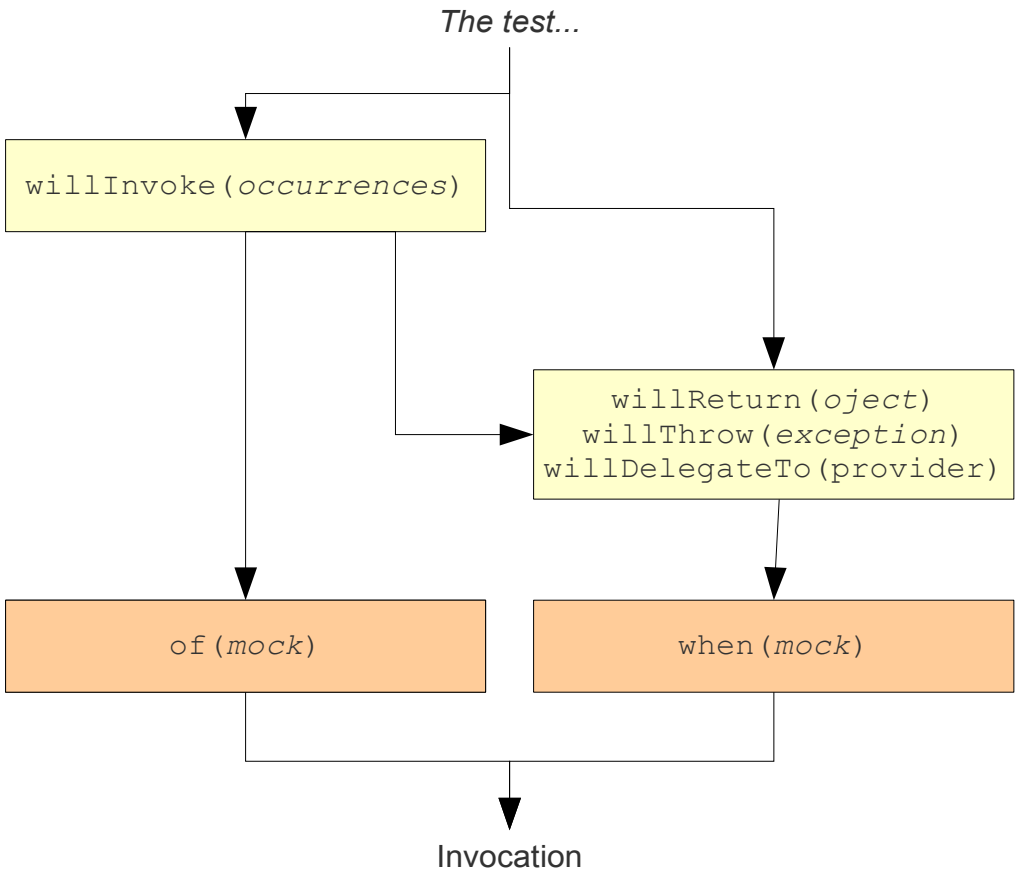
```
Scenario scenario = new Scenario() {
    {
        // Your expectations.
    }
};
Stubs stubs = new Stubs() {
    {
        // Your stubs
    }
};
Story story = Story.create(scenario, stubs);
story.begin();
// Your test
story.end();
```

You simply have to import the schemer and write:

```
begin();
// Your expectations, your stubs, your test
end();
```

You can mix new expectations and stubs with your actual test, or still use a clear section for each part, as you wish...

The directives follow a pretty simple syntax that you may think of as "***the test will...***" (for example, the test will invoke this method this number of times). More synthetically, the syntax of directives complies with the following graph:


The test...
willInvoke(occurrences)
willReturn(oject)
willThrow(exception)
willDelegateTo(provider)
of(mock)
when(mock)
Invocation

This means that you have three possible combinations to create directives:

- Create an expectation for `mock.method(arguments...)` conforming to `occurrences`:
  - `willInvoke(`*`occurrences`*`).of(`*`mock`*`).method(`*`arguments...`*`)`
- Create a stub applying `result` when `mock.method(arguments...)` is invoked:
  - `willReturn|Throw(`*`result`*`).when(`*`mock`*`).method(`*`arguments...`*`)`
- Create an expectation for which the invocation result is specified:
  - `willInvoke(`*`occurrences`*`).willReturn|`
    `Throw(`*`result`*`).when(`*`mock`*`).method(`*`arguments...`*`)`

The following example gives an illustration of masquerades:

```
1. package com.vmware.lmock.example.tutorial;
2.
3. import static com.vmware.lmock.checker.Occurrences.atLeast;
4. import static com.vmware.lmock.masquerade.Schemer.*;
5. import com.vmware.lmock.impl.Mock;
6. …
7. public class MasqueradeExample {
8.     @Test
9.     public void testAList() {
10.             @SuppressWarnings("unchecked")
11.             final List<String> list = Mock.getObject(List.class);
12.
13.             begin();
14.
15.             // The user may verify that the list is empty and add strings.
16.             willReturn(true).when(list).isEmpty();
17.             willInvoke(atLeast(1)).of(list).add(aNonNullOf(String.class));
18.
19.             // Run this portion of test...
20.             if (list.isEmpty()) {
21.                 list.add("string 1");
22.                 list.add("string 2");
23.             }
24.
25.             // Now the user will read back the values.
26.             willInvoke(1).willReturn("string 1").when(list).get(0);
27.             willInvoke(1).willReturn("string 2").when(list).get(1);
28.
29.             assertEquals("string 1", list.get(0));
30.             assertEquals("string 2", list.get(1));
31.
32.             end();
33.         }
34.     }
```

Finally, you can mix masquerades with “traditional” scenarios and stubs, using the `append` clause, hence re-use predefined patterns in your tests.

Tips: class specialization

Lmock implements a pretty complete process regarding the validation of the classes of arguments passed to an invocation. In particular, the argument clauses allow to specialize the class of arguments to subclasses of the initially expected type.

For example, let's consider a list of objects. You can use the argument clauses to impose that add(Object item) explicitly expects strings rather than general objects:

```
1. import com.vmware.lmock.impl.Mock;
2. import com.vmware.lmock.impl.Scenario;
3. import com.vmware.lmock.impl.Story;
4. import static com.vmware.lmock.checker.Occurrences.*;
5.
6. public class ClassSpecializationExample {
7.     @SuppressWarnings("unchecked")
8.     private final List<Object> list = Mock.getObject(List.class);
9.
10.          @Test
11.          public void testAddWithStringsOnly() {
12.              Story story = Story.create(new Scenario() {
13.                  {
14.                      expect(list).add(aNonNullOf(String.class));
15.                      occurs(atLeast(1));
16.                  }
17.              });
18.              story.begin();
19.              list.add("hello world");
20.              list.add(5);
21.              story.end();
22.          }
23.      }
```

The invocation on line 19 will pass, since the argument is a string, but the invocation on line 20 causes an exception, since the argument is an integer:

```
com.vmware.lmock.exception.UnexpectedInvocationError: expectation error: unexpected
invocation of 'Mock(List)$0.add(5)'

what happened up to now:
```

satisfied 1 times: interface java.util.List.add(String!=null):void/[1..]

Tips: array arguments

The strategy of Lmock regarding arrays is the following:

- When an array is an argument of an expectation or a stub, keep the reference to that array
- Whenever the corresponding method is invoked, check each element of the argument with the current contents of the reference

This means that you can adopt different strategies concerning the way you expect such arguments.

The first strategy is to define the expected values as a constant array. Any array that has exactly the same contents than your reference is considered as equal, even if it's not the same object.

The second strategy is to create your own array as reference, so that it can be modified during your

tests. Once again, an argument will be considered as equal to your array as long as its contents are the same.

To illustrate those cases, let's take the following example: a reservoir (represented by the `Reservoir` interface) is filled by a `Supplier` (the class under test):

```
interface Reservoir {
    void fill(String[] data);
}

class Supplier {
    private final Reservoir reservoir;

    Supplier(Reservoir reservoir) {
        this.reservoir = reservoir;
    }

    void supply(String... data) {
        reservoir.fill(data);
    }
}
final Reservoir reservoir = Mock.getObject(Reservoir.class);
```

In the first test, we define the arguments to the expected method supply as a constant array of strings:

```
@Test
public void testArrayWithConstantReference() {
    Story story = Story.create(new Scenario() {
        {
            expect(reservoir).fill(new String[] { "water", "mud" });
        }
    });
```

The test (which passes) consists in requesting the supplier to provide the same data:

```
        story.begin();
        new Supplier(reservoir).supply("water", "mud");
        story.end();
```

The second test uses an external reference for the expected array argument:

```
@Test
public void testArrayWithVariableReference() {
    final String[] reference = new String[] { "water", "mud" };
    Story story = Story.create(new Scenario() {
        {
            expect(reservoir).fill(reference);
        }
    });
```

Once again, if we invoke the supplier with the same data as the contents of reference, the test will pass:

```
        story.begin();
        new Supplier(reservoir).supply("water", "mud");
```

But we may also change the contents of the reference on the fly, thus changing the expectation:

```
reference[1] = "oil";
new Supplier(reservoir).supply("water", "oil");
story.end();
```

Tips: using stubs or expectations in directives

… Yes you can say, for example:

```
1.      @SuppressWarnings("unchecked")
2.      final List<String> list1 = Mock.getObject(List.class);
3.      @SuppressWarnings("unchecked")
4.      final List<String> list2 = Mock.getObject(List.class);
5.
6.      @Test
7.      public void testWithReference() {
8.          begin();
9.          willReturn("hello").when(list1).get(0);
10.         willInvoke(1).of(list2).add(list1.get(0));
11.
12.         list2.add(list1.get(0));
13.         end();
14.           }
```

On line 9, the test defines a stub for the method `list1.get(0)`. The simulation result is used as argument to the specification of an expectation (on line 10).

You can do that... with a slight constraint: **never invoke the mock for which you create a directive in the arguments of the invoked method**.

In other words, the following is forbidden:

```
begin();
willReturn("hello").when(list1).get(0);
willInvoke(1).of(list1).add(list1.get(0));
end();
```

Common mistakes

Mix of `with`, `anyOf`, `aNonNullOf`

Be careful, those three clauses are independent! In other words:

- `with(anyOf(...))` is wrong
- `with(aNonNullOf(...))` is wrong too

Lmock does not raises any error in that case.

Use invocation checkers as arguments to methods

Lmock naively considers that an invocation checker passed as argument to an invocation must be taken as is. For example:

```
....when(list).contains(StringChecker.valuesContain("$")));
```

Means that you expect to check that `list` contains an element of type `StringChecker`.

If you want to express “an argument that matches the checker” don't forget to use an argument clause:

```
....when(list).contains(with(StringChecker.valuesContain("$")));
```