jUnit Testing With Mockito

By Ken Cooney

Intro

This PDF covers how to test your Java application using JUnit and Mockito. I'll assume that you have basic understanding on how to write Java code. The code to test will be insanely simple as the focus on this document is testing and not necessarily about software development in Java.

Setup

If you want to run through the examples, you will need the junit and mockito jars. I used the following jar files for testing the examples in this document:

- junit-4.8.2.jar
- mockito-all.1.9.0.jar
- mockito-core-1.9.5.jar

Junit can be found at http://sourceforge.net/projects/junit/
Mockito can be found at http://code.google.com/p/mockito/

To make the project a bit fun, we'll be simulating a knight going on a quest. I found this example in the book Spring In Action by Gallardo, Burnette, and McGovern. I'll be expanding upon their example to cover various concepts in jUnit and Mockito testing.

Create a new project using your favorite development tool (I prefer Eclipse). You can call the project Sandbox or Knights or whatever you want. Create a package called main.java.app.kobj. We'll put two classes there: Quest and BraveKnight. The next page shows the code for those classes.

Code to Test

===

```
package main.java.app.kobj;

public class Quest
{
	public void embark()
	{
		System.out.println("Embarking on a quest.");
	}
}
```

===

```
package main.java.app.kobj;

public class BraveKnight implements Knight
{
	private Quest quest;

	public BraveKnight(Quest quest)
	{
		this.quest=quest;
	}

	public void embarkOnQuest()
	{
		quest.embark();
	}
}
```

===

The First Insanely Simple Test

Create a package called main.java.app.kobj.test.

In that package, create a class called BraveKnightTest.

The only thing we need to test is if the knight embarks on a quest when you call embarkOnQuest. Since quest is an instance of a class object, we don't want the BraveNightTest to run the quest and do whatever quest does. Granted in this example, Quest just prints something. But what if quest was a webservice that may or may not be up? We just want to verify that we called quest.embark().

So, we're going to mock Quest. This is super simple since quest doesn't return anything.

```
package main.java.app.kobj.test;

import static org.mockito.Mockito.*;
import main.java.app.kobj.*;
import org.junit.Test;
import org.junit.Assert.*;


public class BraveKnightTest
{
	@Test
	public void knightShouldEmbarkOnQuest()
	{
		Quest mockQuest=mock(Quest.class);
	}
}
```

Mock simulates calling the class object but it doesn't do anything. Now, we need to pass the quest to the BraveKnight constructor.

```
		BraveKnight knight=new BraveKnight(mockQuest);
```

Now we want to embark on the quest.

```
		knight.embarkOnQuest();
```

The knight pretends to embark on the quest. He pretends to ride a horse but actually is clapping two halves of a coconut together and trotting around, making neighing sounds. We want to verify he actually embarked on the pretend quest, to make sure embarkOnQuest() called quest.embark();

```
	verify(mockQuest, times(1)).embark();
```

The entire test should look like the following:

```
@Test
public void knightShouldEmbarkOnQuest()
{
	Quest mockQuest=mock(Quest.class);
	BraveKnight knight=new BraveKnight(mockQuest);
	knight.setMotivated(true);
	knight.embarkOnQuest();
	verify(mockQuest, times(1)).embark();
}
```

Now we run the jUnit test. Right click the BraveKnightTest class. Select Run As > JUnit Test. We get a tab with JUnit and a green bar. Did we really test anything? Let's tweak BraveKnight and find out.

```
private Quest quest;
private boolean motivated;

public BraveKnight(Quest quest)
{
	this.quest=quest;
}

public void embarkOnQuest()
{

	if (motivated)
		quest.embark();
}
```

Let's run the test again. The knight is not motivated, so he doesn't embark on the quest. We get a red bar. The test failed.

So indeed, we tested that the quest.embark() method executed. The reason we use Mockito is because it allows us to test our Class (BraveKnight in this example) without testing other class objects. Let's tweak things some more. Add the following to BraveKnight:

```
public void setMotivated(boolean motivated)
{
	this.motivated=motivated;
}
```

Add this to the BraveKnightTest, just before embarkOnQuest.

```
knight.setMotivated(true);
```

Run it. We should get a green bar.

Now create a second test. Let's verify that an unmotivated knight doesn't go on the quest. We want to verify that the embark method was called zero times.

```
@Test
public void knightShouldNotEmbarkOnQuest()
{
	Quest mockQuest=mock(Quest.class);
	BraveKnight knight=new BraveKnight(mockQuest);
	knight.embarkOnQuest();
	verify(mockQuest, times(0)).embark();
}
```

Run it. Both tests should pass. Now we have test coverage and we didn't need to run the quest's embark() method.

Mocking a Method that Returns Primitives

The two previous tests are fine and dandy when they don't return. However, most methods return something. So, let's start with primitives. We'll first do a boolean.

Let's create a boolean in quest to say if the quest has been completed. Add the following to Quest:

```
private boolean completed;
public boolean isCompleted()
{
	return completed;
}
```

No, we're not going to add a setter. We're not testing Quest, we're testing BraveKnight and pretending to check and see if the Quest is done. It's a pretend quest, so the knight can pretend he completed it. So, let's add something in BraveKnight that calls the quest's isCompleted() method. Obviously a quest can't be completed before it is started, so I added that and updated embarkOnQuest.

```
private boolean embarkedOnQuest;

public boolean isQuestDone()
{
	return embarkedOnQuest && quest.isCompleted();
}

public void embarkOnQuest()
{
	if (motivated)
	{
		quest.embark();
		embarkedOnQuest=true;
	}
}
```

The test is insanely simple. We're just returning the boolean. Let's write the new test.

```
@Test
public void knightQuestCompleted()
{
	Quest mockQuest=mock(Quest.class);

	// brave knight, go forth!
	BraveKnight knight=new BraveKnight(mockQuest);
	knight.setMotivated(true);
	knight.embarkOnQuest();
	boolean status=knight.isQuestDone();

	// Test that quest has started but not finished
	verify(mockQuest, times(1)).embark();
	AssertFalse("Should be False (not completed).",status);
}
```

So, we're asserting that status should return false. If the assertion fails, the message "Should be False (not competed)" will show up in the JUnit test bar in addition to the bar being red. This helps us know why the test fails.

But here's the catch. The isQuestDone() calls quest.isCompleted(). How to we pretend to run that call? Mockito has a mechanism that says when you are asked to run a method, return this value. So, we'll have isCompleted() return false. Put this below the Quest constructor:

```
when(mockQuest.isCompleted()).thenReturn(false);
```

Run the test and yes, we get a green bar. This test passed. Now let's test finishing the quest. We'll see why it didn't matter that we didn't create a setComplete method in Quest. We're pretending to call isQuestDone().

```
@Test
public void knightQuestCompleted()
{
	Quest mockQuest=mock(Quest.class);
	when(mockQuest.isCompleted()).thenReturn(true);

	BraveKnight knight=new BraveKnight(mockQuest);
	knight.setMotivated(true);
	knight.embarkOnQuest();
	boolean status=knight.isQuestDone();

	verify(mockQuest, times(1)).embark();
	AssertTrue("Should be True (completed).",status);
}
```

This time we're using assertTrue to tell JUnit that we're expecting status to be true. We haven't done complete code coverage. We need to verify that you can't complete a quest before starting it as well as verifying the scenario where a quest is not started and it's not completed. I'll leave those tests for you to complete.

Now we'll do a test that involves a method of a mocked object that returns an integer. Any Quest worth partaking requires the knight to travel somewhere. So we'll keep track of the distance traversed while doing the quest. Add the following to Quest.

```
int distance;
public int getDistance()
{
	return distance;
}
```

Once again, we don't need a setter since we're not testing Quest. So the king requires an update of how far the knight has travelled. Don't worry, the king has come up with a way to do this and it involves putting a note inside a coconut and having two swallows carry it by the husk and fly to the king's castle. We'll make it simple and not simulate the swallows or the coconut. We'll just add the following method to BraveKnight that gets how many miles the knight travelled.

```
public int getMilesTravelled()
{
	return quest.getDistance();
}
```

Now, there probably should be a way for the knight to update the miles travelled on his quest, but for demonstration purpose, we're making this insanely simple. Let's create the test.

```
@Test
public void knightQuestDistance()
{
	Quest mockQuest=mock(Quest.class);
	when(mockQuest.getDistance()).thenReturn(100);

	BraveKnight knight=new BraveKnight(mockQuest);
	int miles=knight.getMilesTravelled();

	AssertEquals(100, miles);
}
```

So, the mock call to getDistance will return 100. At the end of the test, we'll assert that the knight.getMilesTravelled returned 100. If you run the test, it will pass.

Try changing the assertEquals value to 101 and run the test. You will see the following error in the JUnit tab:

Java.lang.AssertionError: expected <101> but was:<100>
at
main.java.app.kobj.test.BraveKnightTest.knightQuestDistance(BraveKnightTest.java:72)

So, the JUnit test will tell you what was expected as well as the line number it failed on.

Moving right along; each Quest has a name associated by it. But the King has enemies everywhere and tells the Knight to only refer to his Quest by a letter. Let's implement that.

Add the following to Quest:

```
char letter;
public char getQuestLetter()
{
	return letter;
}
```

Add the following the BraveKnight:

```
public char tellQuestLetter()
{
	return quest.getQuestLetter();
}
```

Now we'll add the following test:

```
@Test
public void knightQuestLetter()
{
	Quest mockQuest=mock(Quest.class);
	when(mockQuest.getQuestLetter()).thenReturn('c');

	BraveKnight knight=new BraveKnight(mockQuest);
	char questLetter=knight.tellQuestLetter();

	AssertEquals('c', questLetter);
}
```

Once again, we run it and we get a green bar. Try changing the 'c' to a 'C'. It should fail. The JUnit tab will indicate the ASCII value of the character expected and the ASCII value it got.

Mocking Methods that Return Strings or Nulls or Objects

The Knight must be able to tell the King the Quest he's on. So, we'll add that. Add the following to Quest:

```
String name;
public String getName()
{
	return name;
}
```

Add the following to BraveKnight.

```
public String tellQuestName()
{
	return quest.getName();
}
```

Now create a new test. For Strings, we'll want to use a string compare and assert the String matches the expected value.

```
@Test
public void knightQuestName()
{
	Quest mockQuest=mock(Quest.class);
	when(mockQuest.getName()).thenReturn("Quest for the Holy Grail");

	BraveKnight knight=new BraveKnight(mockQuest);
	String questName = knight.tellQuestName();
	AssertTrue(("Quest for the Holy Grail").equalsIgnoreCase(
		questName));
}
```

Let's pretend a knight doesn't have a quest yet. Strings are initially initialized to null. So, how do we test for that scenario?

```
@Test
public void knightNoQuestName()
{
	Quest mockQuest=mock(Quest.class);
	when(mockQuest.getName()).thenReturn(null);

	BraveKnight knight=new BraveKnight(mockQuest);
	String questName = knight.tellQuestName();
	AssertNull(questName);
}
```

Consequently, if you wanted to verify something was not null:

```
	AssertNotNull(questName);
```

Now, what if we wanted the Knight to have the ability to tell the King everything about the quest? We would make a method that returns the Quest instantiation. This isn't so bad. Add this to BraveKnight:

```
public Quest getQuest()
{
	return quest;
}
```

Now we need to test this.

```
	@Test
	public void knightGetQuest()
	{
		Quest mockQuest=mock(Quest.class);
		when(mockQuest.getName()).thenReturn(null);

		BraveKnight knight=new BraveKnight(mockQuest);
		Quest myQuest = knight.getQuest();

		AssertEquals(myQuest,mockQuest);
	}
```

Let's try something a bit trickier. Make a new quest called DragonQuest that extends Quest. We'll add a new method in a minute. Now this DragonQuest requires the Knight to come back with the dragon's head. So, make a DragonsHead class.

```
package main.java.app.kobj;

public class DragonsHead
{
	String scaleColor;

	public DragonsHead(String scaleColor)
	{
		this.scaleColor=scaleColor;
	}

	String getScaleColor()
	{
		return scaleColor;
	}
}
```

So, let's update DragonQuest to add a getQuestProof() method. For simplicity sake, we won't add logic to test to see if the quest was completed. Feel free to do that and add the test cases that test it. Also, to simplify things, I'll presume the King's DragonQuest requires that dragon's scales to be red.

```
	public DragonsHead getQuestProof()
	{

		return new DragonsHead("red");
	}
```

I'm also going to add a getQuestProof() to Questo BraveKnight won't show an error when we try to get proof of a quest's completion:

```
	public Object getQuestProof()
	{

		return "Proof.";
	}
```

Now we need to update BraveKnight to show proof:

```
	public Object getQuestProof()
	{
		return quest.getQuestProof();
	}
```

Now we need to test that we got proof of the quest is completed. We also want to check to verify if the object is not null.

```
	@Test
	public void knightDragonQuest()
	{
		DragonQuest mockQuest=mock(DragonQuest.class);
		DragonsHead dragonHead = new DragonsHead("red");
		when(mockQuest.getQuestProof()).thenReturn(dragonHead);

		BraveKnight knight=new BraveKnight(mockQuest);
		Object proof = knight.getQuestProof();

		AssertNotNull("Object is null", proof);
		AssertTrue((proof instanceof DragonsHead));
	}
```

Testing For Thrown Exceptions

It is entirely possible that our mocked object may throw an exception when certain situations come up. The catch is the object is mocked. So, how do we tell Mockito that the mocked object's method threw an exception?

First thing we need to do is create a QuestException. We'll make this stupid simple.

```
package main.java.app.kobj;

public class QuestException extends Exception
{
	private static final long serialVersionUID = 1L;

}
```

Now we need a method in the Quest that will throw that exception. Let's pretend the Quest requires the BraveKnight to cross a bridge. If the bridge is out, embark() throws a QuestException since there's no way to cross the great ravine to complete the quest. Once

again, for practice and testing purposes we don't care what embark does and it doesn't even have to throw the exception. We're testing BraveKnight and not Quest. We're going to simulate calling the method and tell the test case that an exception was thrown. Update Quest's embark() method as follows.

```
public void embark() throws QuestException
{
	System.out.println("Embarking on a quest.");
}
```

We'll also have to update BraveKnight's and Knight's embarkOnQuest() method. Make the following change to both classes.

```
public void embarkOnQuest() throws QuestException
```

And finally, we'll need add throws QuestException to any BraveKnightTest methods that indicate errors.

```
public void knightQuestDistance() throws QuestException
```

Now, let's simulate an exception thrown because the bridge has been destroyed. Add the following test to the BraveKnightTest. We don't really need the assertFalse(success), but I'm adding it so we know the test works.

```
@Test
public void knightQuestException() throws QuestException
{

	Quest mockQuest=mock(Quest.class);
	boolean success=false;
	QuestException ex = new QuestException("Thrown by Mockito");
	doThrow(ex).when(mockQuest).embark();
	try
	{
		BraveKnight knight=new BraveKnight(mockQuest);
		knight.setMotivated(true);
		knight.embarkOnQuest();
		success=true;
		assertFalse(success);
	}
	catch (Exception e)
	{
		assertFalse(success);
		assertTrue(e instanceof QuestException);
		assertEquals(ex.getMessage(), "Thrown by Mockito");
	}
}
```

We run the test and it passes. If you comment out the doThrow command and rerun it, you will see the test fail.

Let's test it with no exception thrown. This will pass. Adding the doThrow will cause this test to fail.

```
@Test
public void knightNoQuestException() throws QuestException
{

	DragonQuest mockQuest=mock(DragonQuest.class);
	boolean success=false;
	try
	{
		BraveKnight knight=new BraveKnight(mockQuest);
		knight.setMotivated(true);
		knight.embarkOnQuest();
		success=true;
		assertTrue(success);
	}
	catch (Exception e)
	{
		assertTrue(success);
	}
}
```

JUnit Testing Methods That Don't Use Mocked Objects

Of course, JUnit came out well before Mockito or even EasyMock (which is where the idea started). There are methods that don't do anything with class objects. So, you can test those without mocking a class. So, don't mock a class if you don't need to. Even if your class is a factory that generates a class object, you don't have to mock that object. And never jUnit test a class you are mocking.

That said, there are things you need to test for to get test coverage.

Function coverage – has each function been tested?
Statement coverage – has each node in the program been executed?
Decision coverage – has each edge in the program been executed? (IF and CASE statements)
Condition coverage – has each Boolean sub-expression evaluated both to true and false?
Parameter value coverage – Has the most common parameter values been tested?
Loop coverage – Has every loop executed zero times, once, and more than once?
Boundary testing – Test for range boundaries. If an integer variable can be between one and ten, then test the following values: 0, 1, 2, 9, 10, and 11.

"Oh Mockito, you will never know anything about my code.
I'll never know hard it is to mock things.
Oh no. Mockito, I'll never know."

-- Elton John "Mockito"