How to Mock Static Methods

We have all read it or heard someone talk about it. "Static Methods are Death to Testability". You can read the article here:
Using PowerMock

Let's start off with the basics. To use PowerMock, you'll have to download it from Google Code here:
http://code.google.com/p/powermock/ (or you can add it to your POM if you are using maven). Once PowerMock is added to your project, mocking static methods is actually very easy.

Mocking a Static Method

As in all of my previous tutorials, we're going to do this one via TDD (Test Driven Development). For this example, we are going to assume we have the following utility class:

```java
import java.net.InetAddress;
import java.net.UnknownHostException;

public final class NetworkUtil {
 public static String getLocalHostname() {
  String hostname = "";
  try {
   InetAddress addr = InetAddress.getLocalHost();
   // Get hostname
   hostname = addr.getHostName();
  } catch ( UnknownHostException e ) {
  }
  return hostname;
 }
}
```

As you can see, it returns the human readable version of the machine name. Now there is an issue with this. Chances are that the above code would run fine if we let it run through a unit test regardless of what environment we are in (*nix, Windows, OS X, etc). The issue is, however, we cannot predict the output. One of the key aspects of unit testing is the ability to predict the output of our test based on a given input. In this case, as we move our test from environment to environment, the results of the above method will change. Enter PowerMock.

So the requirements for our use of the above utility class is to create a URL to an image hosted on the local machine. The final URL should be in the following format:

http://LOCAL_MACHINE_NAME/myapplication/images/myimage.gif

Now you can stop laughing at the example. The purpose of this tutorial is not to find some painful testing feature, it's to show where the ability to mock static method calls is useful. So let's start our test:

```java
import org.junit.runner.RunWith;
import org.powermock.core.classloader.annotations.PrepareForTest;

@RunWith( PowerMockRunner.class )
@PrepareForTest( NetworkUtil.class )
public class URLGeneratorTest {
```

}

You'll notice immediately that there are two annotations needed to use PowerMock. First, we don't use the default JUnit runner when we use PowerMock. PowerMock provides it's own runner which we will need to use, specified with the @RunWith annotation. The other annotation we are using @PrepareForTest is used to identify all of the classes that PowerMock will be mocking. In our case, we will be mocking one class, so we pass in only one. If we have multiple, it can take an array of classes as well.

The creation of our setUp and shell test methods should look normal. Nothing exciting here:

```java
import org.junit.runner.RunWith;
import org.powermock.core.classloader.annotations.PrepareForTest;
import static org.powermock.api.easymock.PowerMock.mockStatic;
import static org.powermock.api.easymock.PowerMock.replayAll;
import static org.powermock.api.easymock.PowerMock.verifyAll;
import static org.easymock.EasyMock.expect;

@RunWith( PowerMockRunner.class )
@PrepareForTest( NetworkUtil.class )
public class URLGeneratorTest {
   private URLGenerator generator;

   @Before
   public void setUp() {
      generator = new URLGenerator();
   }

   @Test
   public void testGenerateURL() {

   }
}
```

Now let's focus specifically on the test method. We're essentially going to do XYZ things:

 - Mock the class. Since static methods exist at the class level, we create a partial mock of all static methods with the mockStatic method.

 - Declare our expectations as normal. One of the biggest points about PowerMock is that the API is designed to mesh well with EasyMock. To record our expectations we do exactly as we normally would.

 - Replay our recorded expectations. PowerMock has a nice little method called replayAll. Instead of listing out all of the mock objects we want to replay, this method handles it for us.

 - Execute the method under test. Nothing exciting here.

 - Verify our execution. PowerMock provides the compliment to replayAll with verifyAll.

 - Assert the results. Normal JUnit applies here

```java
   @Test
```

```
    public void testGenerateURL() {
        mockStatic( NetworkUtil.class );

        expect( NetworkUtil.getLocalHostname() ).andReturn( "localhost" );

        replayAll();
        String results = generator.generateURL();
        verifyAll();

        assertEquals(
            "http://localhost/myapplication/images/myimage.gif",
            results );
    }
```

That's it. The above code is all you need to test those "nasty" static methods. The untestable is now testable. Oh, wait. I guess we should write the actual implementing code so we can prove all of this works. Ok, check out the method that implements this test below:

```
public class URLGenerator {
 public String generateURL() {
  return "http://" +
                NetworkUtil.getLocalHostname() +
                "/myapplication/images/myimage.gif";
 }

}
```

I never said it would be pretty. But what it does do is it implements the test correctly and demonstrates my point.

Conclusion

So that's it. Mocking static methods (and other "untestable code") is actually quite easy when you put PowerMock to work. One thing I want to point out that PowerMock helped me see. Just because you can't do something with your current toolset doesn't mean that it's bad. It just means you may need to re-evaluate your toolset.

As always, I'm excited to read your comments! Have you used PowerMock? What about other ways of addressing the same issue? Let's hear 'em!