Testing Private Methods in Java

We've been writing a lot of Java lately, and a lot of tests. We *always* write tests, right? Alas, no cool record-and-playback stuff like FlexMonkey or FoneMonkey, just plain old JUnit 4 tests with plenty of Hamcrest goodness.

Suddenly, I realized that I really needed to test some private methods. So, a quick google for "testing private methods java" brings up a good article by Bill Venners. He lists all possible options to test private methods:

1. Don't test private methods
2. Give the methods package access
3. Use a nested test class
4. Use reflection

Basically, the only real one is #4, use reflection. Bill didn't give me the exact code I needed, so lots of googling later I realized that the world is filled with opinionated people (like me), and boy do they love to talk about #1. I just wanted some code, not a lecture, so I had to write my own code. Here is that code for anyone else that just wants to test private methods.

Imagine you have a class `MyClass` and it has a private method, `myMethod()`. Sorta like this:

```
public class MyClass {
    private String myMethod(String s) {
        return s;
    }
}
```

Then you could use reflection to invoke the method like this:

```
MyClass myClass = new MyClass();
Method method = MyClass.class.getDeclaredMethod("myMethod", String.class);
method.setAccessible(true);
String output = (String) method.invoke(myClass, "some input");
```

The real magic is `setAccessible(true)` which allows the private method to be called outside the class. And shazam, I can now test all my private methods. I was really hoping JUnit 4 would provide some additional facilities specifically for testing private methods, but not luck.

A Better Example

Here's a more complete example. Suppose we have a `NovelWriter` class that in a feat of API cleanliness only exposes the `writeNovel()` method. It happens to have a few private utility methods that we'd like to test:

```
public class NovelWriter {
    public String writeNovel() {
        //...the magic goes here...
        return null;
```

```
    }

    private String shout(String s) {
        return s.toUpperCase().replaceAll("\\.", "!");
    }

    private List<Integer> countLetters(List<String> words) {
        List<Integer> out = new ArrayList<Integer>();
        for (String word : words) {
            out.add( word.replaceAll("[^A-Za-z]+","").length() );
        }
        return out;
    }
}
```

I won't get into all the details, but it seems easy to imagine a clean API that has private helper methods. Furthermore, it seems very logical to me to want to bring all methods, both public and private, so I can be sure they are being exercised to the fullest.

Our JUnit 4 + Hamcrest test class:

```
public class NovelWriterTest {
    public static NovelWriter novelWriter;

    @BeforeClass
    public static void beforeClass() {
        novelWriter = new NovelWriter();
    }

    @Test
    public void privateShout() throws NoSuchMethodException,
            InvocationTargetException, IllegalAccessException {

        String input = "This is magic.";

        Method method = NovelWriter.class.getDeclaredMethod("shout", String.class);
        method.setAccessible(true);
        String output = (String) method.invoke(novelWriter, input);

        assertThat(output, notNullValue());
        assertThat(output, is("THIS IS MAGIC!"));
    }

    @Test
    @SuppressWarnings("unchecked")
    public void privateCountLetters() throws NoSuchMethodException,
            InvocationTargetException, IllegalAccessException {

        List<String> input = Arrays.asList("Foo", "Foobar123", "Foo Bar Baz");

        Method method = NovelWriter.class.getDeclaredMethod("countLetters", List.class);
        method.setAccessible(true);
        List<Integer> output = (List<Integer>) method.invoke(novelWriter, input);

        assertThat(output, notNullValue());
        assertThat(output.size(), is(3));
        assertThat(output, hasItems(3, 6, 9));
    }
}
```

The nice thing about using the Reflection API like this is that it really doesn't get too messy. I'm not inspecting anything at runtime, because I know exactly the return type and the types of all the parameters. I'm just invoking the method with known inputs, followed by a simple cast on the output type. And as you can see in the second test above, `privateCountLetters()`, it's not a problem to use generics because we're not doing any inspection only invocation.

Happy testing.