ABOUT HUGH HAMILL

Hugh is a Senior Software Engineer and Certified Scrum Master based in Galway, Ireland. He achieved his B.Sc. in Applied Computing from Waterford Institute of Technology in 2002 and has been working in industry since then. He has worked for a several large blue chip software companies listed on both the NASDAQ and NYSE.

Custom Hamcrest Matchers

Posted by: Hugh Hamill in Core Java November 15th, 2015

This article is part of our Academy Course titled Testing with Mockito.

In this course, you will dive into the magic of Mockito. You will learn about Mocks, Spies and Partial Mocks, and their corresponding Stubbing behaviour. You will also see the process of Verification with Test Doubles and Object Matchers. Finally, Test Driven Development (TDD) with Mockito is discussed in order to see how this library fits in the concept of TDD. Check it out here!

In this tutorial we will use the Hamcrest API to create our own Custom Matchers, in order to extend the 'out of the box' functionality that Hamcrest provides.

Table Of Contents

1. Why Custom Matchers?

From time to time we will run up against the limits of the Hamcrest library of Matchers. Either we need new matcher functionality on standard classes such as Strings, Integers or Lists, or we need to create matchers which match with highly customized classes which we have created. In this tutorial we will create matchers for both circumstances, using the tools which Hamcrest provides.

2. Anatomy of a Matcher

In order to create our Custom Matchers we will be extending the built in Abstract class

```
TypeSafeDiagnosingMatcher
```

. If you extend this class in your IDE for an Integer type you will see the two abstract methods from this class:

```
public class BlankMatcher extends TypeSafeDiagnosingMatcher<Integer> {
    @Override
    protected boolean matchesSafely(Integer integer, Description description) {
        return false;
    }

    @Override
    public void describeTo(Description description) {

    }
}
```

The first method,

```
matchesSafely()
```

, is where the meat of our Matcher goes, this is the method which is executed by Hamcrest when we want to use our Matcher to test a value. It is also responsible for reporting why the Matcher failed to match, if that is the case. The mismatch description is what is used by Hamcrest after the 'But:' section of it's output for a failing matcher, and so the mismatch description should be formatted accordingly.

The second method,

```
describeTo()
```

, is used to generate a description of what the matcher is checking. This description is what is used by Hamcrest after the 'Expected:' section of its output for a failing matcher, and so the description should be formatted accordingly.

Under the hood this Matcher will check that the input value is not null, and is of the correct type, so we are guaranteed to have a value of the correct type to match against by the time execution hits our

```
matchesSafely
```

method.

Hamcrest does the heavy lifting for us, so these two methods are all we need to implement our own Hamcrest Matchers. In the coming examples we will also add some syntactic sugar to make it easy to create and use our Custom Matchers.

3. Custom Matchers for Existing Classes

In this section we will create a number of Custom matchers that work on existing types.

3.1. isEven()

Let's start by creating a matcher to determine if an input number is an Even number. As before we will extend the

```
TypeSafeDiagnosingMatcher
```

for an Integer.

```
public class IsEven extends TypeSafeDiagnosingMatcher<Integer> {
    @Override
    protected boolean matchesSafely(Integer integer, Description description) {
        return false;
    }

    @Override
    public void describeTo(Description description) {

    }
}
```

We will then implement the

```
describeTo()
```

method to provide a description of what we expect from the matcher:

```java
@Override
public void describeTo(Description description) {
   description.appendText("An Even number");
}
```

We can use the

`description`

parameter to create our Matcher's description. The

`Description`

class provides a number of helper methods for formatting input and in this case we are using the

`appendText()`

method to simply add some text.

Next let's create our error message, using the

`description`

parameter which is passed to the

`matchesSafely`

method. Remember, this output will only be seen under a failure condition. We will use a couple of methods in the

`Description`

class to format our message:

```java
@Override
protected boolean matchesSafely(Integer integer, Description description) {
   description.appendText("was ").appendValue(integer).appendText(", which is an Odd number");
   return false;
}
```

Next we'll implement the actual check on the number to see if it is even or not:

```java
@Override
protected boolean matchesSafely(Integer integer, Description description) {
   description.appendText("was ").appendValue(integer).appendText(", which is an Odd number");
   return integer % 2 == 0;
}
```

Lastly we will add a static factory method to the class to make it easy to use in tests:

```java
public static IsEven isEven() {
   return new IsEven();
}
```

Putting it all together we have the following Matcher class:

```java
public class IsEven extends TypeSafeDiagnosingMatcher<Integer> {
   @Override
   protected boolean matchesSafely(Integer integer, Description description) {
       description.appendText("was ").appendValue(integer).appendText(", which is an Odd number");
       return integer % 2 == 0;
   }

   @Override
   public void describeTo(Description description) {
       description.appendText("An Even number");
   }

   public static IsEven isEven() {
       return new IsEven();
   }
}
```

We are now able to write some tests using our new matcher.

We create a new Test class called

`IsEvenTest`

and then we import our new factory method:

```java
import static com.javacodegeeks.hughwphamill.mockito.hamcrest.matchers.IsEven.isEven;

public class IsEvenTest {

}
```

Next we'll write a test method to test the positive case where the matcher evaluates as true.

```
@Test
public void should_pass_for_even_number() throws Exception {
   // Given
   Integer test = 4;

   // Then
   assertThat(test, isEven());
}
```

And a method to show the output where the matcher fails to match.

```
@Test
public void should_fail_for_odd_number() throws Exception {
   // Given
   Integer test = 5;

   // Then
   assertThat(test, isEven());
}
```

Which will generate the following output:

```
java.lang.AssertionError:
Expected: An Even number
     but: was <5>, which is an Odd number
```

Normally when writing real tests for matchers we want a passing test to ensure that the logic is sound, after all we don't want to work on projects with failing tests! If we wanted to do this we could change the assert to something like the following:

```
assertThat(test, not(isEven()));
```

However it can be useful to write failing tests during development in order to manually check the output from the matcher.

3.2. divisibleBy(Integer divisor)

We have now seen how to create a custom Matcher to test an intrinsic property of an Integer; is it odd or even? However we see many Matchers in the hamcrest Matcher library that test against an input value. We will now create such a Matcher ourselves.

Imagine we want to regularly test numbers to find if they are divisible by another number. We can write a custom Matcher to do this for us.

Let's start by creating a no-arg Matcher like in our last example and hardcode the divisor to 3.

```
public class DivisibleBy extends TypeSafeDiagnosingMatcher<Integer> {

   @Override
   protected boolean matchesSafely(Integer integer, Description description) {
       int remainder = integer % 3; // Hardcoded to 3 for now!

       description.appendText("was ").appendValue(integer)
               .appendText(" which left a remainder of ").appendValue(remainder);
       return remainder == 0;
   }

   @Override
   public void describeTo(Description description) {
       description.appendText("A number divisible by 3"); // Hardcoded to 3 for now!
   }

   public static DivisibleBy divisibleBy() {
       return new DivisibleBy();
   }
}
```

Our Matcher looks quite like our

```
IsEven()
```

Matcher, but with a slightly more complex mismatch description.

How can we change from our hardcoded value to an input value? There's no trick to it, really it's actually as easy as adding a private member variable and setting it in the constructor, then passing in the value through the factory method.

Let's look at the complete Matcher now:

```
public class DivisibleBy extends TypeSafeDiagnosingMatcher<Integer> {

   private final Integer divisor;

   public DivisibleBy(Integer divisor) {
       this.divisor = divisor;
   }

   @Override
   protected boolean matchesSafely(Integer integer, Description description) {
       int remainder = integer % 3; // Hardcoded to 3 for now!

       description.appendText("was ").appendValue(integer)
               .appendText(" which left a remainder of ").appendValue(remainder);
       return remainder == 0;
   }

   @Override
   public void describeTo(Description description) {
        description.appendText("A number divisible by 3"); // Hardcoded to 3 for now!
    }

    public static DivisibleBy divisibleBy(Integer divisor) {
        return new DivisibleBy(divisor);
    }
}
```

Again, let's create a couple of tests to exercise our new Matcher:

```
public class DivisibleByTest {

    @Test
    public void should_pass_for_true_divisor() throws Exception {
        // Given
        Integer test = 15;

        // Then
        assertThat(test, is(divisibleBy(5)));
    }

    @Test
    public void should_fail_for_non_divisor() throws Exception {
        // Given
        Integer test = 17;

        // Then
        assertThat(test, is(divisibleBy(3)));
    }
}
```

The output from the failing test is

```
java.lang.AssertionError:
Expected: is A number divisible by 3
     but: was <17> which left a remainder of <2>
```

We have now seen how to create Custom Matchers for existing classes which can either test intrinsic properties or accept test values.

Next we will create Custom Matchers for classes we have written ourselves.

4. Custom Matchers for Your Own Classes

Custom Hamcrest matchers are a powerful tool in our test arsenal when we are working with Classes we have created ourselves. We will now create a domain model and write some Custom Matchers to work with that model.

4.1. Our Model: A Tree

A common data structure in programming is a Tree made up of many nodes. These nodes may have a single Parent or one or more Children. A node which has no parents is called a Root. A node which has no children is called a Leaf. A node X is considered a Descendant of another node Y if a path can be traced from X to Y through the parent of X. A node X is considered an Ancestor of another node Y if Y is a descendent of X. A node X is considered a Sibling of another node Y if both X and Y share a parent.

We will use our Node to store a single int value.

Our model will have a single class called Node which looks like this:

```
/**
* Node class for building trees
*

* Uses instance equality.
*/
public class Node {

    private final int value;

    private Node parent;
    private final Set<Node> children;

    /**
     * Create a new Node with the input value
     */
    public Node(int value) {
        this.value = value;
        children = new HashSet<>();
    }

    /**
     * @return The value of this Node
     */
    public int value() {
        return value;
    }

    /**
     * @return The parent of this Node
     */
    public Node parent() {
        return parent;
    }

    /**
```

```
     * @return A copy of the Set of children of this Node
     */
    public Set<Node> children() {
        return new HashSet<>(children);
    }

    /**
     * Add a child to this Node
     *
     * @return this Node
     */
    public Node add(Node child) {
        if (child != null) {
            children.add(child);
            child.parent = this;
        }
        return this;
    }

    /**
     * Remove a child from this Node
     *
     * @return this Node
     */
    public Node remove(Node child) {
        if (child != null && children.contains(child)) {
            children.remove(child);
            child.parent = null;
        }
        return this;
    }

    public String toString() {
        StringBuilder builder = new StringBuilder();
        builder.append("Node{")
                .append("value=").append(value).append(",")
                .append("parent=").append(parent != null ?parent.value : "null").append(",")
                .append("children=").append("[")
                .append(children.stream().map(n ->
Integer.toString(n.value)).collect(Collectors.joining(",")))
                .append("]}");

        return builder.toString();
    }
}
```

Note that for this simplistic example our class is not thread safe.

This model allows us to build up a tree of nodes, starting with a root node and adding children. We can see this in action in the following small application class:

```
public class App {

    public static void main(String... args) {
        Node root = createTree();

        printNode(root);
    }

    private static Node createTree() {
        /*
                         1
                     /       \
                   2           3
                 /   \       /   \
               4       5   6       7
             /   \       |
           8      9   10
        */
        Node root = new Node(1);
        root.add(
                new Node(2).add(
                        new Node(4).add(
                                new Node(8)
                        ).add(
                                new Node(9)
                        )

                ).add(
                        new Node(5).add(
                                new Node(10)
                        )
                )
        ).add(
                new Node(3).add(
                        new Node(6)
                ).add(
                        new Node(7)
                )
        );
        return root;
    }

    private static void printNode(Node node) {
        System.out.println(node);
        for (Node child : node.children()) {
            printNode(child);
        }

    }
}
```

Which will produce the following output:

```
Node{value=1,parent=null,children=[3,2]}
Node{value=3,parent=1,children=[7,6]}
Node{value=7,parent=3,children=[]}
Node{value=6,parent=3,children=[]}
Node{value=2,parent=1,children=[5,4]}
Node{value=5,parent=2,children=[10]}
Node{value=10,parent=5,children=[]}
Node{value=4,parent=2,children=[8,9]}
Node{value=8,parent=4,children=[]}
Node{value=9,parent=4,children=[]}
```

Now we have defined our model and know how to use it we can start to create some Matchers against it.

We are going to use a Node test fixture in our classes in order to test against a consistent model, which will be the same tree structure we defined in our example application. The fixture is defined here:

```java
public class NodeTestFixture {

   static Node one = new Node(1);
   static Node two = new Node(2);
   static Node three = new Node(3);
   static Node four = new Node(4);
   static Node five = new Node(5);
   static Node six = new Node(6);
   static Node seven = new Node(7);
   static Node eight = new Node(8);
   static Node nine = new Node(9);
   static Node ten = new Node(10);

   static {
       one.add(two);
       one.add(three);

       two.add(four);
       two.add(five);

       three.add(six);
       three.add(seven);

       four.add(eight);
       four.add(nine);

       five.add(ten);
   }
}
```

4.2. leaf()

The first Matcher we will create will check if the input node is a leaf node. It will accomplish this by checking if the input node has any children, if it does then it is not a leaf node.

```java
public class IsLeaf extends TypeSafeDiagnosingMatcher<Node> {
   @Override
   protected boolean matchesSafely(Node node, Description mismatchDescription) {
       if (!node.children().isEmpty()) {
           mismatchDescription.appendText("a node with ")
                   .appendValue(node.children().size())
                   .appendText(" children");
           return false;
       }
       return true;
   }

   @Override
   public void describeTo(Description description) {
       description.appendText("a leaf node with no children");
   }

   public static IsLeaf leaf() {
       return new IsLeaf();
   }
}
```

Test Class:

```java
public class IsLeafTest extends NodeTestFixture {

   @Test
   public void should_pass_for_leaf_node() throws Exception {
       // Given
       Node node = NodeTestFixture.seven;

       // Then
       assertThat(node, is(leaf()));
   }

   @Test
   public void should_fail_for_non_leaf_node() throws Exception {
       // Given
       Node node = NodeTestFixture.four;

       // Then
       assertThat(node, is(not(leaf())));
   }
}
```

4.3. root()

We will now create a matcher to check if a Node is the Root Node. We will do this by checking for the presence of a parent Node.

```java
public class IsRoot extends TypeSafeDiagnosingMatcher<Node> {
    @Override
    protected boolean matchesSafely(Node node, Description mismatchDescription) {
        if (node.parent() != null) {
            mismatchDescription.appendText("a node with parent ")
                    .appendValue(node.parent());
            return false;
        }
        return true;
    }

    @Override
    public void describeTo(Description description) {
        description.appendText("a root node with no parent");
    }

    public static IsRoot root() {
        return new IsRoot();
    }
}
```

Test Class:

```java
public class IsRootTest {

    @Test
    public void should_pass_for_root_node() throws Exception {
        // Given
        Node node = NodeTestFixture.one;

        // Then
        assertThat(node, is(root()));
    }

    @Test
    public void should_fail_for_non_root_node() throws Exception {
        // Given
        Node node = NodeTestFixture.five;

        // Then
        assertThat(node, is(not(root())));
    }
}
```

4.4. descendantOf (Node node)

Next up is a matcher with input, we will check if a given Node is a descendant of an input Node. We will move up through the parents to see if we hit the test Node before a Root.

```java
public class IsDescendant extends TypeSafeDiagnosingMatcher<Node> {

    private final Node ancestor;

    public IsDescendant(Node ancestor) {
        this.ancestor = ancestor;
    }

    @Override
    protected boolean matchesSafely(Node node, Description description) {
        while (node.parent() != null) {
            if (node.parent().equals(ancestor)) {
                return true;
            }
            node = node.parent();
        }
        description.appendText("a Node which was not a descendant of ")
                .appendValue(ancestor);
        return false;
    }

    @Override
    public void describeTo(Description description) {
        description.appendText("a descendant Node of ").appendValue(ancestor);
    }

    public static IsDescendant descendantOf(Node ancestor) {
        return new IsDescendant(ancestor);
    }
}
```

Test Class:

```java
public class IsDescendantTest {

    @Test
    public void should_pass_for_descendant_node() throws Exception {
        // Given
        Node node = NodeTestFixture.nine;
        Node ancestor = NodeTestFixture.two;

        // Then
        assertThat(node, is(descendantOf(ancestor)));
    }
    @Test
    public void should_fail_for_non_descendant_node() throws Exception {
        // Given
        Node node = NodeTestFixture.ten;
        Node ancestor = NodeTestFixture.three;

        // Then
        assertThat(node, is(not(descendantOf(ancestor))));
    }
}
```

4.5. ancestorOf (Node node)

Next will check if a given Node is an ancestor of an input Node. This operation is effectively the opposite of

```
descendantOf()
```

so we will move up the parents of the input Node instead of the test Node.

```
public class IsAncestor extends TypeSafeDiagnosingMatcher<Node> {

    private final Node descendant;

    public IsAncestor(Node descendant) {
        this.descendant = descendant;
    }

    @Override
    protected boolean matchesSafely(Node node, Description description) {
        Node descendantCopy = descendant;
        while (descendantCopy.parent() != null) {
            if (descendantCopy.parent().equals(node)) {
                return true;
            }
            descendantCopy = descendantCopy.parent();
        }
        description.appendText("a Node which was not an ancestor of ")
                .appendValue(descendant);
        return false;
    }

    @Override
    public void describeTo(Description description) {
        description.appendText("an ancestor Node of ").appendValue(descendant);
    }

    public static IsAncestor ancestorOf(Node descendant) {
        return new IsAncestor(descendant);
    }
}
```

Test Class:

```
public class IsAncestorTest {
    @Test
    public void should_pass_for_ancestor_node() throws Exception {
        // Given
        Node node = NodeTestFixture.two;
        Node descendant = NodeTestFixture.ten;

        // Then
        assertThat(node, is(ancestorOf(descendant)));
    }

    @Test
    public void should_fail_for_non_ancestor_node() throws Exception {
        // Given
        Node node = NodeTestFixture.three;
        Node descendant = NodeTestFixture.eight;

        // Then
        assertThat(node, is(not(ancestorOf(descendant))));
    }
}
```

4.6. siblingOf (Node node)

Lastly we will create a Matcher to check if an input Node is a sibling of another node. We will check if they share a parent. In addition we will do some checks and provide some outputs for when the user tries to test root nodes for siblings.

```
public class IsSibling extends TypeSafeDiagnosingMatcher<Node> {

    private final Node sibling;

    public IsSibling(Node sibling) {
        this.sibling = sibling;
    }

    @Override
    protected boolean matchesSafely(Node node, Description description) {
        if (sibling.parent() == null) {
            description.appendText("input root node cannot be tested for siblings");
            return false;
        }

        if (node.parent() != null && node.parent().equals(sibling.parent())) {
```

```java
            return true;
        }

        if (node.parent() == null) {
            description.appendText("a root node with no siblings");
        }
        else {
            description.appendText("a node with parent ").appendValue(node.parent());
        }

        return false;
    }

    @Override
    public void describeTo(Description description) {
        if (sibling.parent() == null) {
            description.appendText("a sibling of a root node");
        } else {
            description.appendText("a node with parent ").appendValue(sibling.parent());
        }
    }

    public static IsSibling siblingOf(Node sibling) {
        return new IsSibling(sibling);
    }
}
```

Test Class:

```java
public class IsSiblingTest {

    @Test
    public void should_pass_for_sibling_node() throws Exception {
        // Given
        Node a = NodeTestFixture.four;
        Node b = NodeTestFixture.five;

        // Then
        assertThat(a, is(siblingOf(b)));
    }

    @Test
    public void should_fail_for_testing_root_node() throws Exception {
        // Given
        Node a = NodeTestFixture.one;
        Node b = NodeTestFixture.six;

        // Then
        assertThat(a, is(not(siblingOf(b))));
    }

    @Test
    public void should_fail_for_input_root_node() throws Exception {
        // Given
        Node a = NodeTestFixture.five;
        Node b = NodeTestFixture.one;

        // Then
        assertThat(a, is(not(siblingOf(b))));
    }

    @Test
    public void should_fail_for_non_sibling_node() throws Exception {
        // Given
        Node a = NodeTestFixture.five;
        Node b = NodeTestFixture.six;

        // Then
        assertThat(a, is(not(siblingOf(b))));
    }
}
```

5. Conclusion

We have now seen how to create Custom Hamcrest Matchers to test both pre-existing standard Java classes and our own classes. In the next tutorial we will put everything we have learned so far together as we learn about a technique of software development that puts Mocking and Testing front and centre; Test Driven Development.

Tagged with: MOCKITO TESTING