**The Hamcrest Tutorial**

[**https://code.google.com/archive/p/hamcrest/wikis/Tutorial.wiki**](https://code.google.com/archive/p/hamcrest/wikis/Tutorial.wiki)

**Introduction**

Hamcrest is a framework for writing matcher objects allowing 'match' rules to be defined declaratively. There are a number of situations where matchers are invaluble, such as UI validation, or data filtering, but it is in the area of writing flexible tests that matchers are most commonly used. This tutorial shows you how to use Hamcrest for unit testing.

When writing tests it is sometimes difficult to get the balance right between overspecifying the test (and making it brittle to changes), and not specifying enough (making the test less valuable since it continues to pass even when the thing being tested is broken). Having a tool that allows you to pick out precisely the aspect under test and describe the values it should have, to a controlled level of precision, helps greatly in writing tests that are "just right". Such tests fail when the behaviour of the aspect under test deviates from the expected behaviour, yet continue to pass when minor, unrelated changes to the behaviour are made.

**My first Hamcrest test**

We'll start by writing a very simple JUnit 3 test, but instead of using JUnit's assertEquals methods, we use Hamcrest's assertThatconstruct and the standard set of matchers, both of which we statically import:

``` import static org.hamcrest.MatcherAssert.assertThat; import static org.hamcrest.Matchers.\*;

import junit.framework.TestCase;

public class BiscuitTest extends TestCase { public void testEquals() { Biscuit theBiscuit = new Biscuit("Ginger"); Biscuit myBiscuit = new Biscuit("Ginger"); assertThat(theBiscuit, equalTo(myBiscuit)); } } ```

The assertThat method is a stylized sentence for making a test assertion. In this example, the subject of the assertion is the object biscuit that is the first method parameter. The second method parameter is a matcher for Biscuit objects, here a matcher that checks one object is equal to another using the Object equals method. The test passes since the Biscuit class defines an equals method.

If you have more than one assertion in your test you can include an identifier for the tested value in the assertion:

assertThat("chocolate chips", theBiscuit.getChocolateChipCount(), equalTo(10)); assertThat("hazelnuts", theBiscuit.getHazelnutCount(), equalTo(3));

**Other test frameworks**

Hamcrest has been designed from the outset to integrate with different unit testing frameworks. For example, Hamcrest can be used with [JUnit](http://junit.org/) 3 and 4 and [TestNG](http://testng.org/). (For details have a look at the examples that come with the full Hamcrest distribution.) It is easy enough to migrate to using Hamcrest-style assertions in an existing test suite, since other assertion styles can co-exist with Hamcrest's.

Hamcrest can also be used with mock objects frameworks by using adaptors to bridge from the mock objects framework's concept of a matcher to a Hamcrest matcher. For example, [JMock](http://jmock.org/) 1's **constraints** are Hamcrest's matchers. Hamcrest provides a JMock 1 adaptor to allow you to use Hamcrest matchers in your JMock 1 tests. JMock 2 doesn't need such an adaptor layer since it is designed to use Hamcrest as its matching library. Hamcrest also provides adaptors for [EasyMock](http://easymock.org/) 2. Again, see the Hamcrest examples for more details.

**A tour of common matchers**

Hamcrest comes with a library of useful matchers. Here are some of the most important ones.

* Core
  + anything - always matches, useful if you don't care what the object under test is
  + describedAs - decorator to adding custom failure description
  + is - decorator to improve readability - see "Sugar", below
* Logical
  + allOf - matches if all matchers match, short circuits (like Java &&)
  + anyOf - matches if any matchers match, short circuits (like Java ||)
  + not - matches if the wrapped matcher doesn't match and vice versa
* Object
  + equalTo - test object equality using Object.equals
  + hasToString - test Object.toString
  + instanceOf, isCompatibleType - test type
  + notNullValue, nullValue - test for null
  + sameInstance - test object identity
* Beans
  + hasProperty - test JavaBeans properties
* Collections
  + array - test an array's elements against an array of matchers
  + hasEntry, hasKey, hasValue - test a map contains an entry, key or value
  + hasItem, hasItems - test a collection contains elements
  + hasItemInArray - test an array contains an element
* Number
  + closeTo - test floating point values are close to a given value
  + greaterThan, greaterThanOrEqualTo, lessThan, lessThanOrEqualTo - test ordering
* Text
  + equalToIgnoringCase - test string equality ignoring case
  + equalToIgnoringWhiteSpace - test string equality ignoring differences in runs of whitespace
  + containsString, endsWith, startsWith - test string matching

**Sugar**

Hamcrest strives to make your tests as readable as possible. For example, the is matcher is a wrapper that doesn't add any extra behavior to the underlying matcher. The following assertions are all equivalent:

assertThat(theBiscuit, equalTo(myBiscuit)); assertThat(theBiscuit, is(equalTo(myBiscuit))); assertThat(theBiscuit, is(myBiscuit));

The last form is allowed since is(T value) is overloaded to return is(equalTo(value)).

**Writing custom matchers**

Hamcrest comes bundled with lots of useful matchers, but you'll probably find that you need to create your own from time to time to fit your testing needs. This commonly occurs when you find a fragment of code that tests the same set of properties over and over again (and in different tests), and you want to bundle the fragment into a single assertion. By writing your own matcher you'll eliminate code duplication and make your tests more readable!

Let's write our own matcher for testing if a double value has the value NaN (not a number). This is the test we want to write:

public void testSquareRootOfMinusOneIsNotANumber() { assertThat(Math.sqrt(-1), is(notANumber())); }

And here's the implementation:

``` package org.hamcrest.examples.tutorial;

import org.hamcrest.Description; import org.hamcrest.Factory; import org.hamcrest.Matcher; import org.hamcrest.TypeSafeMatcher;

public class IsNotANumber extends TypeSafeMatcher {

@Override public boolean matchesSafely(Double number) { return number.isNaN(); }

public void describeTo(Description description) { description.appendText("not a number"); }

@Factory public static Matcher notANumber() { return new IsNotANumber(); }

} ```

The assertThat method is a generic method which takes a Matcher parameterized by the type of the subject of the assertion. We are asserting things about Double values, so we know that we need a Matcher<Double>. For our Matcher implementation it is most convenient to subclass TypeSafeMatcher, which does the cast to a Double for us. We need only implement the matchesSafely method - which simply checks to see if the Double is NaN - and the describeTo method - which is used to produce a failure message when a test fails. Here's an example of how the failure message looks:

assertThat(1.0, is(notANumber()));

fails with the message

java.lang.AssertionError: Expected: is not a number got : <1.0>

The third method in our matcher is a convenience factory method. We statically import this method to use the matcher in our test:

``` import static org.hamcrest.MatcherAssert.assertThat; import static org.hamcrest.Matchers.\*;

import static org.hamcrest.examples.tutorial.IsNotANumber.notANumber;

import junit.framework.TestCase;

public class NumberTest extends TestCase {

public void testSquareRootOfMinusOneIsNotANumber() { assertThat(Math.sqrt(-1), is(notANumber())); } } ```

Even though the notANumber method creates a new matcher each time it is called, you should not assume this is the only usage pattern for your matcher. Therefore you should make sure your matcher is stateless, so a single instance can be reused between matches.

**Sugar generation**

If you produce more than a few custom matchers it becomes annoying to have to import them all individually. It would be nice to be able to group them together in a single class, so they can be imported using a single static import much like the Hamcrest library matchers. Hamcrest helps out here by providing a way to do this by using a **generator**.

First, create an XML configuration file listing all the Matcher classes that should be searched for factory methods annotated with the org.hamcrest.Factory annotation. For example:

```

```

Second, run the org.hamcrest.generator.config.XmlConfigurator command-line tool that comes with Hamcrest. This tool takes the XML configuration file and generates a single Java class that includes all the factory methods specified by the XML file. Running it with no arguments will display a usage message. Here's the output for the example.

``` // Generated source. package org.hamcrest.examples.tutorial;

public class Matchers {

public static org.hamcrest.Matcher is(T param1) { return org.hamcrest.core.Is.is(param1); }

public static org.hamcrest.Matcher is(java.lang.Class param1) { return org.hamcrest.core.Is.is(param1); }

public static org.hamcrest.Matcher is(org.hamcrest.Matcher param1) { return org.hamcrest.core.Is.is(param1); }

public static org.hamcrest.Matcher notANumber() { return org.hamcrest.examples.tutorial.IsNotANumber.notANumber(); }

} ```

Finally, we can update our test to use the new Matchers class.

``` import static org.hamcrest.MatcherAssert.assertThat;

import static org.hamcrest.examples.tutorial.Matchers.\*;

import junit.framework.TestCase;

public class CustomSugarNumberTest extends TestCase {

public void testSquareRootOfMinusOneIsNotANumber() { assertThat(Math.sqrt(-1), is(notANumber())); } } ```

Notice we are now using the Hamcrest library is matcher imported from our own custom Matchers class.