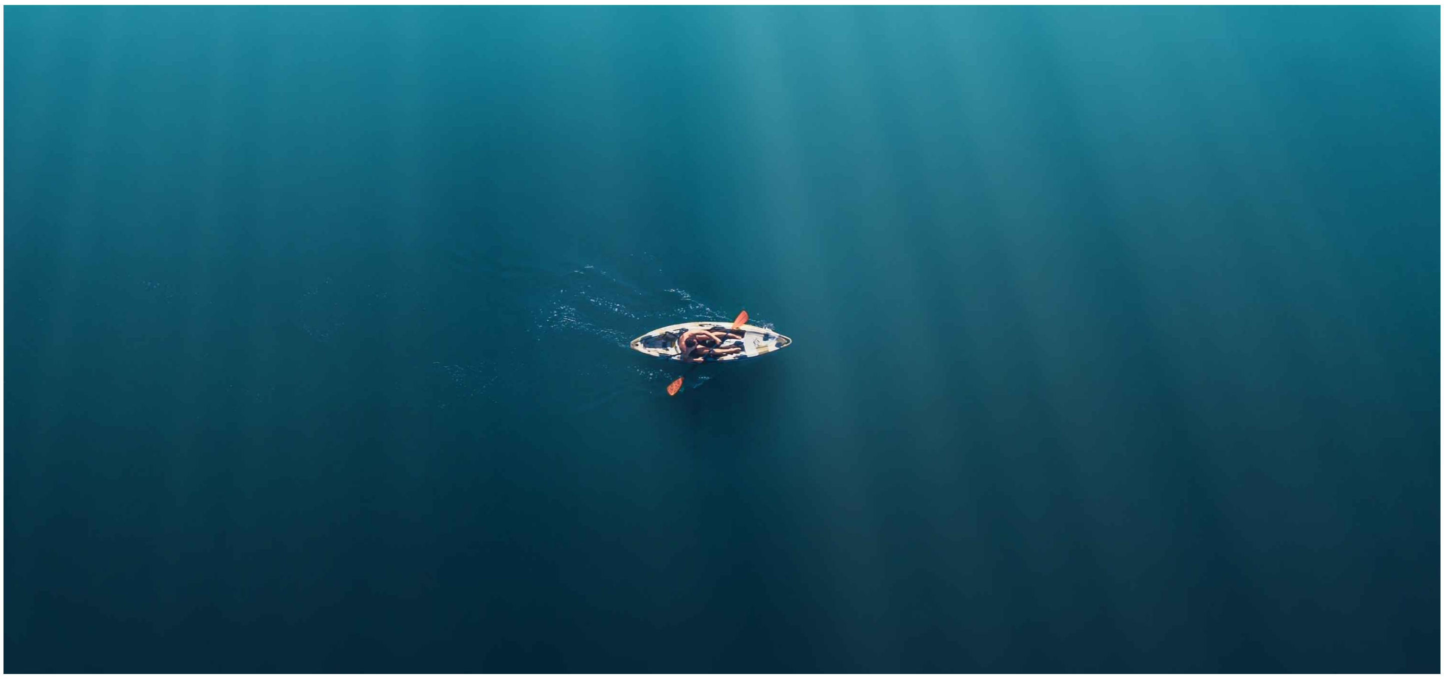

42-Hamcrest

Author: Vincent Lau

Learning Objectives

- Understand the purpose of using testing framework Hamcrest
- Able to code Hamcrest style assertions
- Bonus: Create custom Hamcrest Matcher

Introduction

Hamcrest is a framework for writing matcher objects in Java, used primarily for testing and specifically for making test assertions **more expressive and readable**. It provides a set of matchers that allow you to define custom criteria for checking whether the actual value being tested meets your expected conditions.

Matchers in Hamcrest **follow a fluent, natural language style, making it easier for developers to write and understand their test assertions**. Instead of relying solely on the default equality checks provided by JUnit's `assertEquals` or `assertThat` methods, **you can use Hamcrest matchers to create custom assertions that read like plain English sentences.**

For example, instead of writing:

```java
// Junit
assertEquals(expectedValue, actualValue);
```

You can use Hamcrest to write:

```java
// Hamcrest
assertThat(actualValue, is(equalTo(expectedValue)));
```

The Hamcrest matchers provide a wide range of options for composing complex assertions and handling various types of data.

Maven Dependency

```xml
<!-- https://mvnrepository.com/artifact/org.hamcrest/hamcrest -->
<dependency>
    <groupId>org.hamcrest</groupId>
    <artifactId>hamcrest</artifactId>
    <version>2.2</version>
    <scope>test</scope>
</dependency>
```

Hamcrest with JUnit

JUnit is used in conjunction with Hamcrest when writing and running unit tests in Java. JUnit provides the testing framework, including annotations like `@Test` for defining test methods and the infrastructure for running tests. Hamcrest, on the other hand, enhances the readability and expressiveness of your test assertions.

While JUnit provides its own set of assertions, you can use Hamcrest alongside JUnit to enhance the readability and expressiveness of your test assertions. Hamcrest's matchers can be integrated with JUnit to create more descriptive and flexible test cases.

To use Hamcrest with JUnit:

1. Include both the JUnit and Hamcrest libraries in your project.

2. Import the necessary Hamcrest classes and methods in your test classes to use Hamcrest matchers.
3. Combine JUnit's testing structure (e.g., `@Test` annotations) with Hamcrest matchers to write expressive assertions.

Here's an example of a JUnit test class that uses Hamcrest matchers:

```java
import static org.hamcrest.MatcherAssert.assertThat;
import static org.hamcrest.Matchers.*;
import org.junit.jupiter.api.Test;

public class MyTest {

    // Junit test framework
    @Test
    void testWithHamcrestMatchers() {
        int actualValue = 42;
        // Using Hamcrest matchers for assertions
        assertThat(actualValue, equalTo(42));
        assertThat(actualValue, greaterThan(30));
        assertThat("Hello, World", containsString("Hello"));
    }
}
```

In this example, JUnit is used for structuring the test (`@Test` annotation), while Hamcrest matchers are used for making assertions. Both libraries work together to create comprehensive and readable test cases.

Key features of Hamcrest

Readable Matchers

Hamcrest provides a set of predefined matchers for common data types and structures. These matchers use a fluent and human-readable syntax, making it easier to express the conditions you want to assert, such as Collection Matchers.

Custom Matchers

You can create your own custom matchers when the built-in ones are not sufficient for your specific testing needs. This allows you to write highly specialized assertions tailored to your application.

Integration with Testing Frameworks

Hamcrest can be integrated with various testing frameworks, including JUnit, TestNG, and others. This allows you to use Hamcrest matchers within your test methods to enhance the readability of your test assertions.

Extensible

Hamcrest is designed to be extensible, allowing you to create custom matchers and compose complex assertions from simpler ones.

Core Matchers

equalTo()

- The `equalTo` matcher tests whether two objects are equal according to their `equals` method.

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

public class CoreMatchersExample {

    public static void main(String[] args) {
        String expectedValue = "Hello";
        String actualValue = "Hello";

        assertThat(actualValue, equalTo(expectedValue));
    }
}
```

is()

- The `is` matcher is an alias for `equalTo`. It provides a more readable alternative for simple equality checks.

```java
assertThat(actualValue, is(expectedValue));
```

not()

- The `not` matcher is used to test the inverse of a given matcher. It negates the condition specified by the matcher.

```java
String actualValue = "World";
assertThat(actualValue, not(equalTo("Hello")));
```

nullValue()

- The `nullValue` matcher tests whether a value is `null`.

```java
String nullString = null;
assertThat(nullString, nullValue());
```

notNullValue()

- The `notNullValue` matcher tests whether a value is not `null`.

```java
String nonNullString = "Hello";
assertThat(nonNullString, notNullValue());
```

These core matchers are the foundation of Hamcrest and are often used in combination with other matchers to create more complex assertions. They provide a clean and expressive way to make basic assertions about the equality and nullity of values in your test cases.

Object Matchers

sameInstance()

The `sameInstance` matcher in Hamcrest is used to assert that two references point to the same object in memory. In other words, it checks if two objects are identical in terms of memory reference, not just their content or equality as determined by the `equals` method.

Here's how you can use the `sameInstance` matcher:

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

public class SameInstanceExample {

    public static void main(String[] args) {
        String str1 = "hello";
        String str2 = "hello";
        // Both references point to the same object
        assertThat(str2, sameInstance(str1));
    }
}
```

If `str2` did not refer to the same object as `str1`, the assertion would fail because `sameInstance` checks for reference equality.

Logical Matchers

In Hamcrest, logical matchers allow you to combine multiple matchers into more complex assertions. Two commonly used logical matchers are `allOf` and `anyOf`. These matchers help you build compound conditions for your assertions

allOf()

The `allOf` matcher checks that all of its component matchers must evaluate to true for the overall assertion to succeed. In other words, it performs a logical AND operation on the provided matchers.

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

public class AllOfExample {

    public static void main(String[] args) {
        int value = 42;

        assertThat(value, allOf(
            greaterThan(30),
            lessThan(50)
        ));
    }
}
```

In this example, the assertion succeeds because `value` is both greater than 30 and less than 50.

anyOf()

The `anyOf` matcher checks that at least one of its component matchers must evaluate to true for the overall assertion to succeed. It performs a logical OR operation on the provided matchers.

Here's an example of using `anyOf`:

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

public class AnyOfExample {

    public static void main(String[] args) {
        int value = 42;

        assertThat(value, anyOf(
            equalTo(30),
            equalTo(42),
            equalTo(60)
        ));
    }
}
```

In this example, the assertion succeeds because `value` is equal to 42, which matches one of the provided matchers.

These logical matchers are valuable when you need to create more complex assertions by combining simpler conditions. You can use them to express compound conditions in a clear and readable manner, making your test cases more expressive and easier to understand.

Comparison Matchers

Comparison matchers in Hamcrest allow you to perform numerical comparisons in your assertions. Here are some commonly used comparison matchers in Hamcrest:

greaterThan

The `greaterThan` matcher checks whether a value is greater than a specified value.

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

public class GreaterThanExample {
    public static void main(String[] args) {
        int value = 42;
        assertThat(value, greaterThan(30));
    }
}
```

In this example, the assertion succeeds because `value` is greater than 30.

greaterThanOrEqualTo

The `greaterThanOrEqualTo` matcher checks whether a value is greater than or equal to a specified value.

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

public class GreaterThanOrEqualToExample {
    public static void main(String[] args) {
        int value = 42;
        assertThat(value, greaterThanOrEqualTo(42));
    }
}
```

In this example, the assertion succeeds because `value` is equal to 42, which meets the greater-than-or-equal-to condition.

lessThan

The `lessThan` matcher checks whether a value is less than a specified value.

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

public class LessThanExample {
    public static void main(String[] args) {
        int value = 42;
        assertThat(value, lessThan(50));
    }
}
```

In this example, the assertion succeeds because the `value` is less than 50.

lessThanOrEqualTo

The `lessThanOrEqualTo` matcher checks whether a value is less than or equal to a specified value.

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

public class LessThanOrEqualToExample {
    public static void main(String[] args) {
        int value = 42;
        assertThat(value, lessThanOrEqualTo(42));
    }
}
```

In this example, the assertion succeeds because `value` is equal to 42, which meets the less-than-or-equal-to condition.

These comparison matchers are useful when you need to verify numerical properties in your test cases. They allow you to express conditions related to greater than, greater than or equal to, less than, and less than or equal to with ease and readability.

Text Matchers

Text matchers in Hamcrest allow you to perform text-based assertions on strings.

Here are some commonly used text matchers in Hamcrest:

startsWith

The `startsWith` matcher checks whether a string starts with a specified prefix.

In this example, the assertion succeeds because the `text` string starts with "Hello."

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

public class StartsWithExample {
    public static void main(String[] args) {
        String text = "Hello, World";
        assertThat(text, startsWith("Hello"));
    }
}
```

endsWith

The `endsWith` matcher checks whether a string ends with a specified suffix.

In this example, the assertion succeeds because the `text` string ends with "World."

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

public class EndsWithExample {
    public static void main(String[] args) {
        String text = "Hello, World";

        assertThat(text, endsWith("World"));
    }
}
```

containsString

The `containsString` matcher checks whether a string contains a specified substring.

In this example, the assertion succeeds because the `text` string contains the substring "Hello."

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

public class ContainsStringExample {
    public static void main(String[] args) {
        String text = "Hello, World";
        assertThat(text, containsString("Hello"));
    }
}
```

emptyString

The `emptyString` matcher checks whether a string is empty (has zero length).

In this example, the assertion succeeds because `emptyText` is an empty string.

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

public class EmptyStringExample {
    public static void main(String[] args) {
        String emptyText = "";
        assertThat(emptyText, emptyString());
    }
}
```

These text matchers are valuable for verifying the content and structure of strings in your test cases. They allow you to express conditions related to string prefixes, suffixes, substrings, and emptiness in a clear and readable manner.

Collection Matchers

Collection matchers in Hamcrest allow you to perform assertions on collections, lists, arrays, and other types of collections. Here are some commonly used collection matchers in Hamcrest:

hasItem

The `hasItem` matcher checks whether a collection contains a specific element.

In this example, the assertion succeeds because `myList` contains the element "banana."

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

public class HasItemExample {
    public static void main(String[] args) {
        List<String> myList = Arrays.asList("apple", "banana", "cherry");
        assertThat(myList, hasItem("banana"));
    }
}
```

hasItems

The `hasItems` matcher checks whether a collection contains specific elements.

In this example, the assertion succeeds because `myList` contains both "apple" and "banana."

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

public class HasItemsExample {
    public static void main(String[] args) {
        List<String> myList = Arrays.asList("apple", "banana", "cherry");
        assertThat(myList, hasItems("apple", "banana"));
    }
}
```

hasSize

The `hasSize` matcher checks the size (number of elements) of a collection.

In this example, the assertion succeeds because `myList` has a size of 3.

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

public class HasSizeExample {
    public static void main(String[] args) {
        List<String> myList = Arrays.asList("apple", "banana", "cherry");
        assertThat(myList, hasSize(3));
    }
}
```

contains

The `contains` matcher checks whether a collection contains a specific list of elements in the same order.

In this example, the assertion succeeds because `myList` contains the elements in the specified order.

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

public class ContainsExample {
    public static void main(String[] args) {
        List<String> myList = Arrays.asList("apple", "banana", "cherry");
        assertThat(myList, contains("apple", "banana", "cherry"));
    }
}
```

containsInAnyOrder

The `containsInAnyOrder` matcher checks whether a collection contains a specific list of elements regardless of their order.

In this example, the assertion succeeds because `myList` contains all the elements, even though their order is different.

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

public class ContainsInAnyOrderExample {
    public static void main(String[] args) {
        List<String> myList = Arrays.asList("apple", "banana", "cherry");
        assertThat(myList, containsInAnyOrder("cherry", "apple", "banana"));
    }
}
```

empty

The `empty` matcher checks whether a collection is empty (has no elements).

In this example, the assertion succeeds because `emptyList` is indeed empty.

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

public class EmptyExample {
    public static void main(String[] args) {
        List<String> emptyList = Collections.emptyList();
        assertThat(emptyList, empty());
    }
}
```

These collection matchers allow you to make specific assertions about the contents, size, and structure of collections in your test cases, making your tests more precise and readable.

Array Matchers

Array matchers in Hamcrest allow you to perform assertions on arrays. Here are some commonly used array matchers in Hamcrest:

arrayContaining

The `arrayContaining` matcher checks whether an array contains the specified elements, in the same order.

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

```

```java
public class ArrayContainingMatcherExample {
    public static void main(String[] args) {
        String[] myArray = {"apple", "banana", "cherry"};
        assertThat(myArray, arrayContaining("apple", "banana", "cherry"));
    }
}
```

arrayContainingInAnyOrder

The `arrayContainingInAnyOrder` matcher checks whether an array contains the specified elements, regardless of their order.

In this example, the assertion succeeds because `myArray` contains all the specified elements, and their order does not matter.

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

public class ArrayContainingInAnyOrderMatcherExample {
    public static void main(String[] args) {
        String[] myArray = {"apple", "banana", "cherry"};
        assertThat(myArray, arrayContainingInAnyOrder("banana", "cherry",
"apple"));
    }
}
```

These array matchers are useful when you need to make specific assertions about the contents and order of elements in arrays. They allow you to express conditions related to arrays in a clear and readable manner in your test cases.

Number Matchers

Number matchers in Hamcrest allow you to perform assertions on numeric values. One commonly used number matcher is `closeTo`.

closeTo

The `closeTo` matcher checks whether a numeric value is close to a specified target value within a given delta (**tolerance**). It's often used when comparing floating-point or double values where exact equality may not be expected due to rounding errors.

Here's how you can use the `closeTo` matcher:

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

public class CloseToMatcherExample {

    public static void main(String[] args) {
        double tolerance = 0.01d;
        double actual = 3.14149d; // 3.13150 <= x <= 3.15149
        assertThat(actual, closeTo(3.13150, tolerance));
        assertThat(actual, not(closeTo(3.13149, tolerance)));
        assertThat(actual, closeTo(3.15149, tolerance));
        assertThat(actual, not(closeTo(3.15150, tolerance)));
    }
}
```

The assertion succeeds because `actualValue` (3.14159) is close to `expectedValue` (3.14) within the specified delta (0.01).

The `closeTo` matcher is useful when you need to perform approximate comparisons of numeric values, allowing for a small margin of error. It helps you avoid precision issues when working with floating-point or double values in your tests.

Type Matchers

Type matchers in Hamcrest allow you to perform assertions based on the types of objects. Two commonly used type matchers are `instanceOf` and `typeCompatibleWith`.

instanceOf

The `instanceOf` matcher checks whether an object is an instance of a specified class or type. It's used to verify the exact class of an object.

In this example, the assertion succeeds because `obj` is an instance of the `String` class.

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

public class InstanceOfMatcherExample {
    public static void main(String[] args) {
        Object obj = "Hello, World";
        assertThat(obj, instanceOf(String.class));
    }
}
```

typeCompatibleWith

The `typeCompatibleWith` matcher checks whether a class is compatible with a specified class or type. It's used to verify that a class is assignable to the specified class or a subclass of it.

In this example, the assertion succeeds because the `Integer` class is compatible with the `Number` class. This means `Integer` is either the `Number` class itself or a subclass of it.

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

public class TypeCompatibleWithMatcherExample {
    public static void main(String[] args) {
        assertThat(Integer.class, typeCompatibleWith(Number.class));
    }
}
```

These type matchers are valuable when you need to verify the types of objects or classes in your test cases. They help ensure that objects are of the expected types and that class hierarchies are appropriately structured.

Custom Matchers

Creating a custom matcher in Hamcrest allows you to define your own assertion conditions for specific types of objects. To create a custom matcher, you'll need to create a class that implements the `Matcher` interface provided by Hamcrest. Here are the steps to create a custom matcher:

Create a Custom Matcher Class

Start by creating a new Java class that implements the `Matcher` interface. The `Matcher` interface has two main methods that you need to implement: `matches` and `describeTo`.

`matches`: This method defines the condition that the object being tested must satisfy. It returns `true` if the object matches the condition, otherwise `false`.

`describeTo`: This method provides a human-readable description of the matcher. It's used to generate failure messages when assertions fail.

Here's an example of a custom matcher that checks if a string contains only uppercase letters:

```java
import org.hamcrest.Description;
import org.hamcrest.Matcher;
import org.hamcrest.TypeSafeMatcher;
```

```java

public class UppercaseStringMatcher extends TypeSafeMatcher<String> {

    @Override
    protected boolean matchesSafely(String item) {
        // Define the condition: Check if the string contains only uppercase letters
        return item.matches("[A-Z]+");
    }

    @Override
    public void describeTo(Description description) {
        // Used to generate failure messages when assertions fail
        description.appendText("a string containing only uppercase letters");
    }

    // Static factory method for creating the matcher
    public static Matcher<String> containsOnlyUppercase() {
        return new UppercaseStringMatcher();
    }
}

```

Use the Custom Matcher in Your Tests

Now, you can use the static `containsOnlyUppercase` method to create instances of the `UppercaseStringMatcher` and make your assertions cleaner:

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

public class CustomMatcherExample {

    public static void main(String[] args) {
        String uppercaseString = "HELLO";
        String mixedCaseString = "HelloWorld";

        // Use the static factory method to create the matcher and perform assertions
        assertThat(uppercaseString, containsOnlyUppercase());
        assertThat(mixedCaseString, not(containsOnlyUppercase()));
    }
}
```

Relationship between Hamcrest and JUnit

Integration with JUnit

Hamcrest can be used alongside JUnit or other testing frameworks. In fact, JUnit 4 and later versions have built-in support for Hamcrest matchers. You can use Hamcrest matchers in your JUnit tests to make your assertions more expressive and human-readable.

JUnit Matchers

JUnit provides its own set of built-in matchers for making assertions, but Hamcrest is often used to enhance the expressive power of JUnit assertions. The `assertThat` method in JUnit is commonly used in conjunction with Hamcrest matchers to create more descriptive and understandable test assertions.

Custom Matchers

Hamcrest allows you to create custom matchers, which can be particularly useful when dealing with complex or custom objects in your tests. These custom matchers can be seamlessly integrated into your JUnit test cases.

Summary

Hamcrest is a library for defining and using custom matchers to make your test assertions more human-readable and expressive. While it can be used independently of JUnit, it is commonly used in combination with JUnit to improve the quality and clarity of test code.