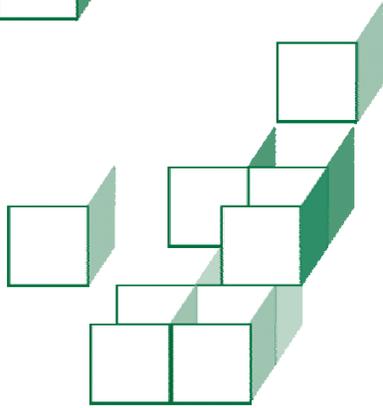
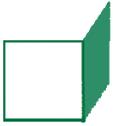




CHAPTER 3

MATCHERS



OBJECTIVES

After completing “Matchers,” you will be able to:

- Describe the advantages of encapsulating verification logic in **matchers**.
- Use built-in matchers from the Hamcrest library.
- Create custom matchers by
 - Aggregating existing ones
 - Implementing **BaseMatcher<T>** or another framework class, directly
 - Building a general-purpose, “functional-friendly” matcher, and then using that with ad-hoc lambda expressions

Hamcrest Matchers

- We've discussed the advantages of reusing test logic, and explored a few strategies for getting that reuse.
- Originally a third-party library, the **Hamcrest** system of **matchers** has been incorporated into JUnit in recent versions.
- A matcher encapsulates the logic involved in verifying specific post-conditions, so that it can be used in multiple test cases.
- While preserving the direct use of various assertion methods such as **assertEquals** and **assertTrue**, JUnit now offers a gateway to Hamcrest matchers, via the **assertThat** methods:

```
public static <T> void assertThat  
    (T actual, Matcher<T> matcher);  
public static <T> void assertThat  
    (String reason, T actual, Matcher<T> matcher);
```

- The Hamcrest “core” is included in the JUnit distribution.
- Extended libraries can be included in a project as well, and we'll work with a JAR that gives us access to everything. The equivalent Maven dependency would be:

```
<dependency>  
    <groupId>org.hamcrest</groupId>  
    <artifactId>hamcrest-all</artifactId>  
    <version>1.3</version>  
</dependency>
```

Advantages of Matchers

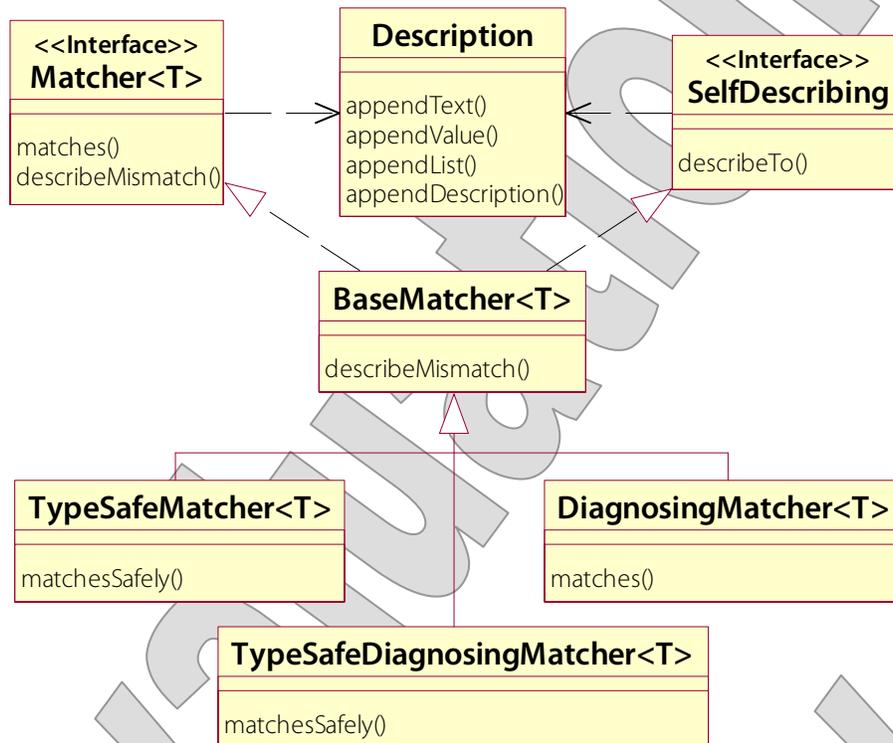
- So the **Matcher<T>** becomes the most natural point of reuse for verification logic, and that's a powerful thing all by itself.
- There are built-in matchers, and you can create your own.
- Its design is also geared toward aggregation and decoration of one matcher by another.
 - A **combining matcher** can apply boolean logic over multiple delegate matchers: all of these must be true, or at least one, etc.
 - A **collection matcher** can apply some other matcher to each of its items, and again we can insist that all match, or at least one, etc.
- Hamcrest offers a “fluent” design that leads to more naturally readable code:

```
assertThat (value, both (greaterThan (9))  
    .and (lessThan (100)));  
assertThat (list, everyItem  
    (hasProperty ("ID", greaterThan (0))));
```

- A matcher encapsulates both matching logic and the description of what's expected and what may have failed to match.
 - This is at first a little inscrutable, and can seem like more of an extra chore than an advantage.
 - But ultimately it's a win to be able to package description logic along with matching logic, especially once we start aggregating and decorating as described above.

The Matcher API

- The following diagram represents what we might call the kernel of the Hamcrest system:



- **Matcher<T>** and **SelfDescribing** are separate starting points, and though it's not immediately obvious they each contribute part of the semantics of error description.
 - **describeTo** is called as part of an **assertThat** execution, when there is a failure – specifically to describe what was **expected**.
 - **describeMismatch** is then called to explain what was wrong with the **actual** value.
 - Between them, they compose the error message, sharing a data record known as a **Description**.

The Matcher API

- The real center of the system is **BaseMatcher<T>** - and in fact they go to some lengths to discourage the programmer from implementing **Matcher<T>** directly.
 - Many matchers will extend this class, and then simply implement **matches** and **describeTo**.
 - It covers **describeMismatch**, with a default message, “was *value*”, which you may override if you wish.
 - **TypeSafeMatcher<T>** is a further refinement that implements **matches** to check for a non-null value; check that the value is of the expected type **T**; and then call the helper method **matchesSafely**, which will be implemented more specifically by the subclass.
 - A **DiagnosingMatcher** just re-shapes the programming model so that you implement one method for both matching and description; and **TypeSafeDiagnosingMatcher<T>** does something similar while also doing null and type-safe checking.

Built-In Matchers

- Then, not shown in the prior diagram, there are plenty of specific matchers that let you apply simple to moderately complex tests against an expected value.
- Though you can use these directly – instantiate a new **IsCollectionWithSize** matcher, and then use it – they routinely expose **factory methods** as well, which can be easier to use and result in more readable code.
- These methods are annotated as **@Factory** methods.
 - This has no real impact on your code.
 - But it lets Hamcrest, as part of its built, gather these methods into a newly-generated utility class, called **Matchers**, which then serves as the one-stop shop for all of these factory methods.
- Here is a sample of the matchers available, shown by their factory methods – many of which have multiple overloads:

`equalTo`
`greaterThan[OrEqualTo]`
`lessThan[OrEqualTo]`
`isIn/isOneOf`
`nullValue/notNullValue`

`instanceOf`

`allOf`
`anyOf`
`both`
`either`
`not`

`containsString`
`startsWith`
`endsWith`

`emptyArray`
`arrayWithSize`
`arrayContaining`

`emptyCollection`
`collectionWithSize`
`collectionContaining`

Collection, Bean, and XPath Matchers

- There are more involved matchers for specific types.
 - We see the string-specific ones on the previous page.
 - For arrays and collections, we can test size and seek out items; we can also assert the success of other, delegate matchers, either requiring that every item pass a test or that at least one item exist that passes the test:

```
everyItem (notNullValue ())  
hasItem (endsWith (".edu"))
```

- We can check contents of maps, too:

```
hasKey ("root")  
hasKey ("root")  
hasValue (lessThan (0))
```

- We can check a JavaBean's properties – individually for existence or for a specific value, or wholesale for equivalence:

```
hasProperty ("ID")  
hasProperty ("ID", equalTo (6))  
samePropertyValuesAs (templateObject)
```

- XML content, represented as a DOM/JAXP **Node**, can be tested using the **XPath** addressing language:

```
hasXPath ("/Record/@Encoding")  
hasXPath ("/Record/@Encoding",  
    startsWith ("ISO-"))
```

- In **Hamcrest_Step1**, a single JUnit test exercises a number of built-in matchers. See **test/cc/test/Hamcrest.java**.

– We start out with some string matching:

```
@Test
public void testHamcrestMatchers()
    throws Exception
{
    String goodBoy = "Every good boy does fine.";

    assertThat (goodBoy, isa (String.class));
    assertThat (goodBoy, startsWith ("Ev"));
    assertThat (goodBoy, containsString ("boy"));
    assertThat (goodBoy, endsWith ("fine."));
```

– We can assert multiple conditions, and join them logically:

```
assertThat (goodBoy, both (startsWith
    ("Every")).and (endsWith (".")));
assertThat (goodBoy, either (isEmptyString
    ()).or (containsString ("boy")));
```

– In case you were wondering, yes, the inequality tests use the full power of Java's **Comparable** system; so for example you can treat chronological relationships as less-than/greater-than:

```
Date earlier = new Date (1471001168000L);
Date later = new Date (1471001169000L);
assertThat (earlier, lessThan (later));
```

- We exercise some of the collection matchers, checking for various properties of a list of numbers in descending order:

```
List<Integer> numbers = new ArrayList<>();
Collections.addAll (numbers, 5, 4, 3, 2, 1);

assertThat (numbers, hasSize (5));
assertThat (numbers, hasItem (is (5)));
assertThat (numbers,
    everyItem (greaterThan (0)));
assertThat (numbers,
    everyItem (lessThan (10)));
assertThat (numbers,
    containsInAnyOrder (1, 2, 3, 4, 5));
assertThat (numbers,
    anyOf (hasItem (1), hasItem (-1)));
```

- And, similarly, with a map:

```
Map<String, Double> probabilities =
    new HashMap<>();
probabilities.put ("Heads", .5001);
probabilities.put ("Tails", .4999);

assertThat (probabilities, hasKey ("Heads"));
assertThat (probabilities, hasValue (.5001));
assertThat (probabilities.get ("Heads") +
    probabilities.get ("Tails"),
    closeTo (1.00, 0.0001));
```

- Notice, by the way, how little explanation the code needs? The fluent-API approach has this advantage, that the code is largely self-explanatory: “assert that every item in the collection is less than 10,” and so on.
- Finally, we use the JavaBeans matcher **samePropertyValuesAs** to compare two objects:

```
MyBean expected = new MyBean ("one", 1);
MyBean actual = new MyBean ("one", 1);
assertThat (actual,
    samePropertyValuesAs (expected));

MyBean firstDelegate =
    new MyBean ("delegate", 1000);
MyBean first = new MyBean ("main", 100);
first.setDelegate (firstDelegate);

// MyBean secondDelegate = ...
MyBean second = new MyBean ("main", 100);
second.setDelegate (firstDelegate);

assertThat (first, is (second));
assertThat (first,
    samePropertyValuesAs (second));
}
```

- Note however a limitation of this matcher: it **doesn't recurse** through object references, so as to test for equivalence, but only **tests for identity** – that is, checks to see that the two references are to the same instance.
- In other words, if you were to plug in **secondDelegate** as the delegate for the **second** bean, the test would fail.

CustomMatcher

- You can build your own matchers to encapsulate application-specific matching logic.
- One approach to this is simply to extend **BaseMatcher<T>** or **TypeSafeMatcher<T>**, and implement the **matches[Safely]** and **describeTo** methods.
- Another is to use the **CustomMatcher<T>** as a base class.
 - This is optimized slightly to support the common case in which the expectation description doesn't require procedural logic, but really is just represented by a simple string.
 - A constructor accepts that string, and **describeTo** is implemented for you to append it to the error message.
 - Then, you only have to implement **matches**.
 - This is meant to support implementation via anonymous classes, as in this example from the Javadoc:

```
Matcher<String> aNonEmptyString =  
    new CustomMatcher<String>("a non empty string") {  
        public boolean matches(Object object) {  
            return ((object instanceof String) &&  
                !((String) object).isEmpty());  
        }  
    };
```

- So that a test could assert as follows:

```
assertThat (myString, aNonEmptyString);
```

Aggregating Matchers

- Another approach is to use one of the combining matchers to assemble more complex matching logic.
- The result can be captured in a simple helper or utility method, and reused easily:

```
public static Matcher<MyBean> hasNameAndNumber ()
{
    return both (hasProperty ("name"))
        .and (hasProperty ("number"));
}
```

– So, in a test method, someone could say ...

```
assertThat (myBean, hasNameAndNumber ());
```

- Again, Hamcrest is designed for progressive decoration, so your new matcher(s) can be used to create others:

```
public static Matcher<List<MyBean>>
    allHaveNameAndNumber ()
{
    return everyItem (hasNameAndNumber ());
}
```

– And this would support ...

```
assertThat (myBeans, allHaveNameAndNumber ());
```

- In `Hamcrest_Step2`, see `test/cc/test/PerfectSquare.java`.

```
public class PerfectSquare
    extends TypeSafeMatcher<Number>
{
    public boolean matchesSafely (Number number)
    {
        double value = number.doubleValue ();
        long root = (long) Math.sqrt (value);
        return root * root == value;
    }

    public void describeTo (Description description)
    {
        description.appendText ("perfect square");
    }

    public static PerfectSquare perfectSquare ()
    {
        return new PerfectSquare ();
    }
}
```

- So, this matcher can test any **Number** or primitive-type number, to see if it is a perfect square.
- We offer a factory method **perfectSquare** which can be used in favor of direct instantiation, though we allow either usage.

- In `test/cc/test/Hamcrest.java`, see a new last line in the `testHamcrest` method:

```
@Test
public void testHamcrestMatchers()
    throws Exception
{
    ...
    assertThat (9, perfectSquare ());
}
```

- This test succeeds, along with all of the others.
- But, try varying the actual value – change 9 to 8 for example – and see how the matcher responds:

```
java.lang.AssertionError:
  Expected: hasProperty("name")
  but: was null
  ...
```

- The “Expected:” content is the result of our `describeTo` implementation, and the “but: was” part is the result of `BaseMatcher<T>`’s default `describeMismatch`.

Suggested time: 30 minutes

In this lab you will build a JUnit test for a component that can fetch bank accounts from a persistent store. In fact the class under test is a false implementation, using hard-coded data, but it provides correct results as would a real persistence component when tested over a prepared database, and the point of this lab will be to write tests for that correct behavior. This will provide an opportunity to explore built-in Hamcrest matchers.

Detailed instructions are found at the end of the chapter.

Evaluated Only

Functional-Friendly Matchers

- Often, there is a need to define a bit of matching logic on the fly, and as of Java 8 we have a number of new tools for injecting a bit of behavior into a bigger algorithm or process – namely, **lambda expressions** and **method references**.
- Currently, Hamcrest's design does not directly support this functional programming style.
 - The **Matcher<T>** interface is not a functional interface.
 - Even if it were, it is not supposed to be implemented directly: we work by subclassing **BaseMatcher<T>** or a subclass thereof.
- Still, it is not too hard to adapt the current design to functional programming.
 - A **BaseMatcher<T>** or **TypeSafeMatcher<T>** subclass could accept matching and/or descriptive logic via functional interfaces.
 - **matches** is basically a **Predicate<T>**, in that it accepts an object and returns a boolean.
 - **describeTo** is a **Consumer<Description>** – though we might just simplify this and take a simple string, the way **CustomMatcher** does.
 - A class that accepts a predicate and either a description-consumer or a string could then act as a general-purpose matcher that would expect to be supplied with matching logic via an anonymous class, lambda expression, or method reference.
- Future versions of Hamcrest may well facilitate this, as **CustomMatcher** already does for anonymous classes.

- Let's add a second custom matcher to this project that will serve as a gateway to a more free-form use of functional programming when defining specific matching criteria.
 - Do your work in **Hamcrest_Step2**.
 - The completed demo is found in **Hamcrest_Step3**.

1. Create a new class **cc.test.FunctionalMatcher**:

```
public class FunctionalMatcher<T>
{
}
```

2. Make it extend **TypeSafeMatcher<T>**:

```
public class FunctionalMatcher<T>
    extends TypeSafeMatcher<T>
{
}
```

3. Give it fields to capture the matching logic and the description:

```
private Predicate<T> test;
private String text;
```

4. Define a constructor that accepts parameters for each of these, and initializes the fields:

```
public FunctionalMatcher
    (Predicate<T> test, String text)
{
    this.test = test;
    this.text = text;
}
```

5. Implement **matchesSafely** by invoking the given predicate:

```
public boolean matchesSafely (T object)
{
    return test.test (object);
}
```

6. Implement **describeTo** by appending the given string:

```
public void describeTo (Description description)
{
    description.appendText (text);
}
```

7. Define a factory method:

```
public static <T> FunctionalMatcher<T> exhibits
    (Predicate<T> test, String text)
{
    return new FunctionalMatcher<T> (test, text);
}
```

8. In `test/cc/test/Hamcrest.java`, at the bottom of `testHamcrest`, apply the new matcher to a number by supplying matching criteria in a simple lambda expression:

```
assertThat (100, exhibits
    ((v) -> v > 99 && v < 1000,
    "three-digit number"));
```

9. Run the class as a JUnit test again, and see that it still passes.
10. Vary the actual value – to 10 or 1000 – and see that it (correctly) fails:

```
java.lang.AssertionError:
  Expected: three-digit number
  but: was <1000>
  ...
```

11. Try the same matcher on a different type of value, with different criteria:

```
assertThat (new Date (259200000000L), exhibits
    ((Date v) -> v.getTime () % 86400000 == 0,
    "midnight on any date"));
```

12. Try it out again, and if you like try varying the actual value to see the error message.
- The matcher is ready for a wide range of types and matching logic, which, for relatively simple criteria, can be supplied in a simple form.

SUMMARY

- Matchers can be a bit inscrutable at first, but once understood they can be a great tool for organizing your test code.
- You can get a good bit of mileage out of the build-in matchers, especially by using the combining matchers on the fly.
- You can also create custom matchers, by a few techniques.
- Note that there are other matcher libraries, and Hamcrest is designed to support this sort of extensibility.
 - We'll see **Mockito** later in the course, which is primarily a dynamic-mocking library, but offers its own set of matchers as well.