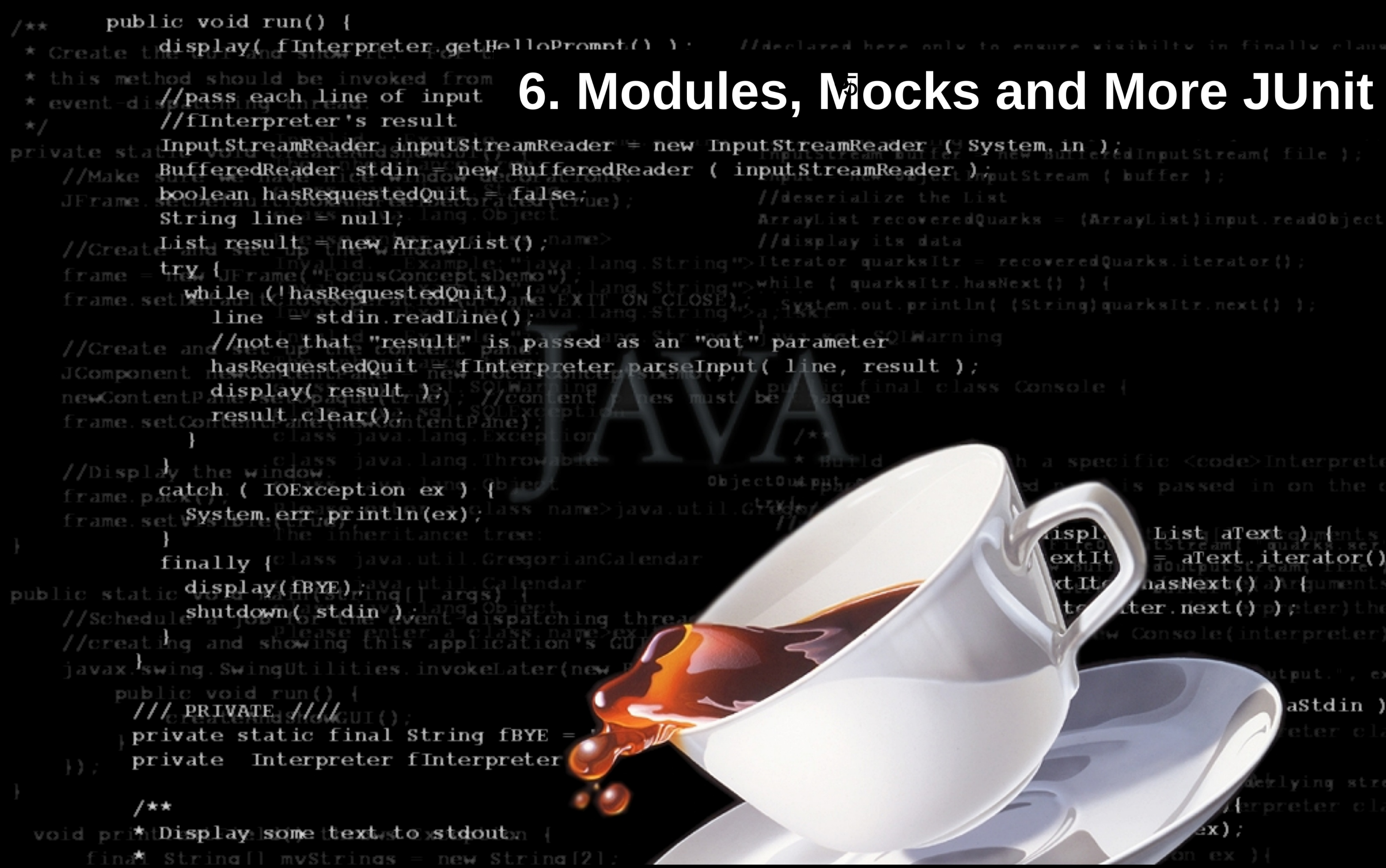


6. Modules, Mocks and More JUnit

Java Test Driven Development with JUnit

Module Topics

1 AssertJ Matchers

2 Jmockit Mocking Library

3 Command Line JUnit

AssertJ Matchers

Third Generation Matchers

- The assertions seen so far are called second generation
  - *First generation was the basic Java assert function*
  - *Second generation is the set of JUnit assertions we have been using*
  - *Third generation are libraries like AssertJ and Hamcrest*
- AssertJ tries to improve on JUnit assertions by:
  - *Expressing assertions in a more natural syntax*
  - *Providing more powerful kinds of assertions*
  - *Making it easier to work with complex object like lists and maps*
  - *Defining a syntax for writing complex assertions*

AssertJ Library

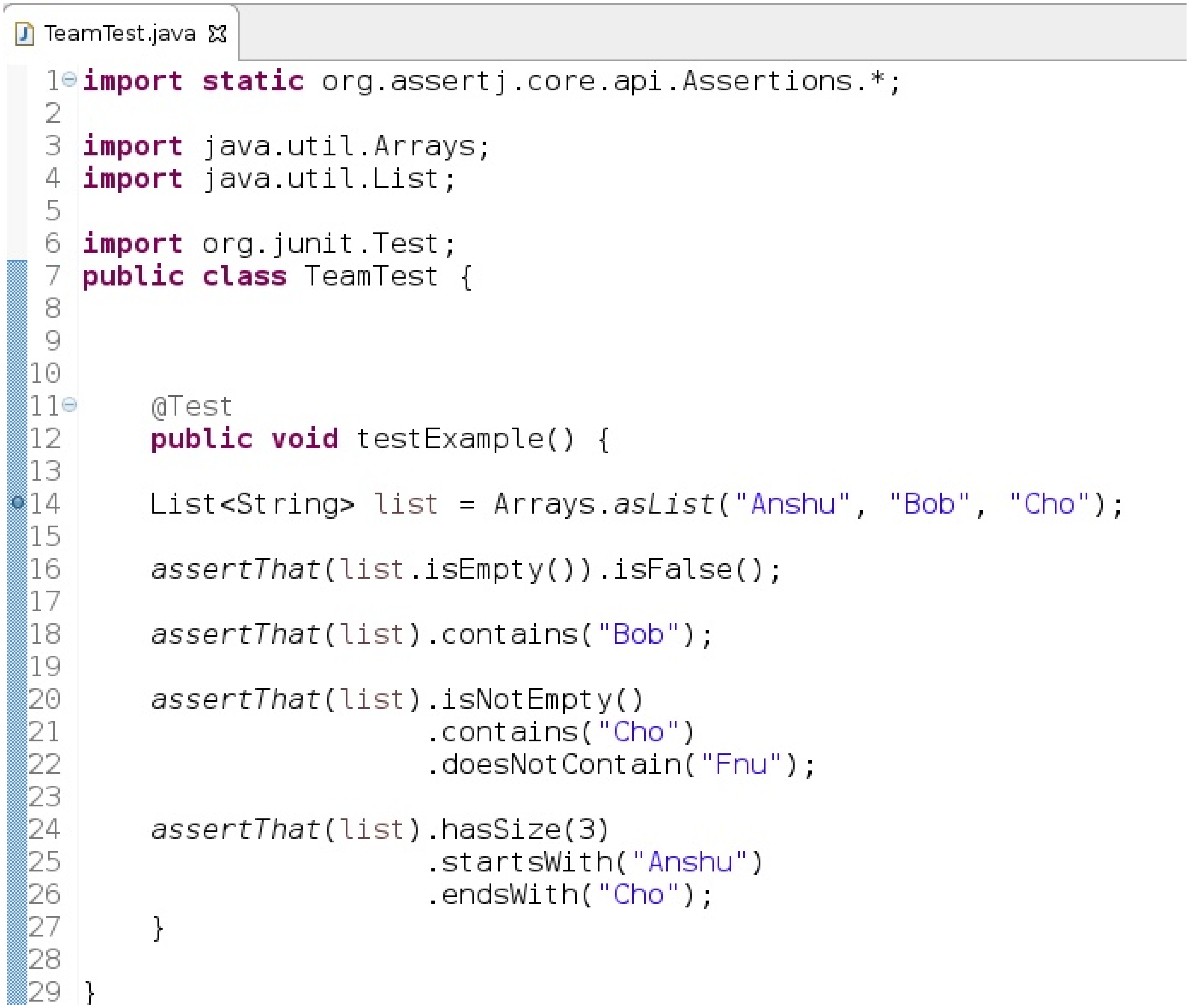
- AsertJ assertions always look like
  - `assertThat(thingToTest).test1().test2()...testn();`
  - *The test conditions are applied one after another*

- We could express a BankAccount assertion as
  - `assertThat(b.getBalance()).isEqualTo(100);`

- This syntax allows for more complex and powerful tests
  - *AssertJ makes testing the structure of complex objects (lists etc) easier*
  - *Using JUnit assertions, we have to write the code ourselves*

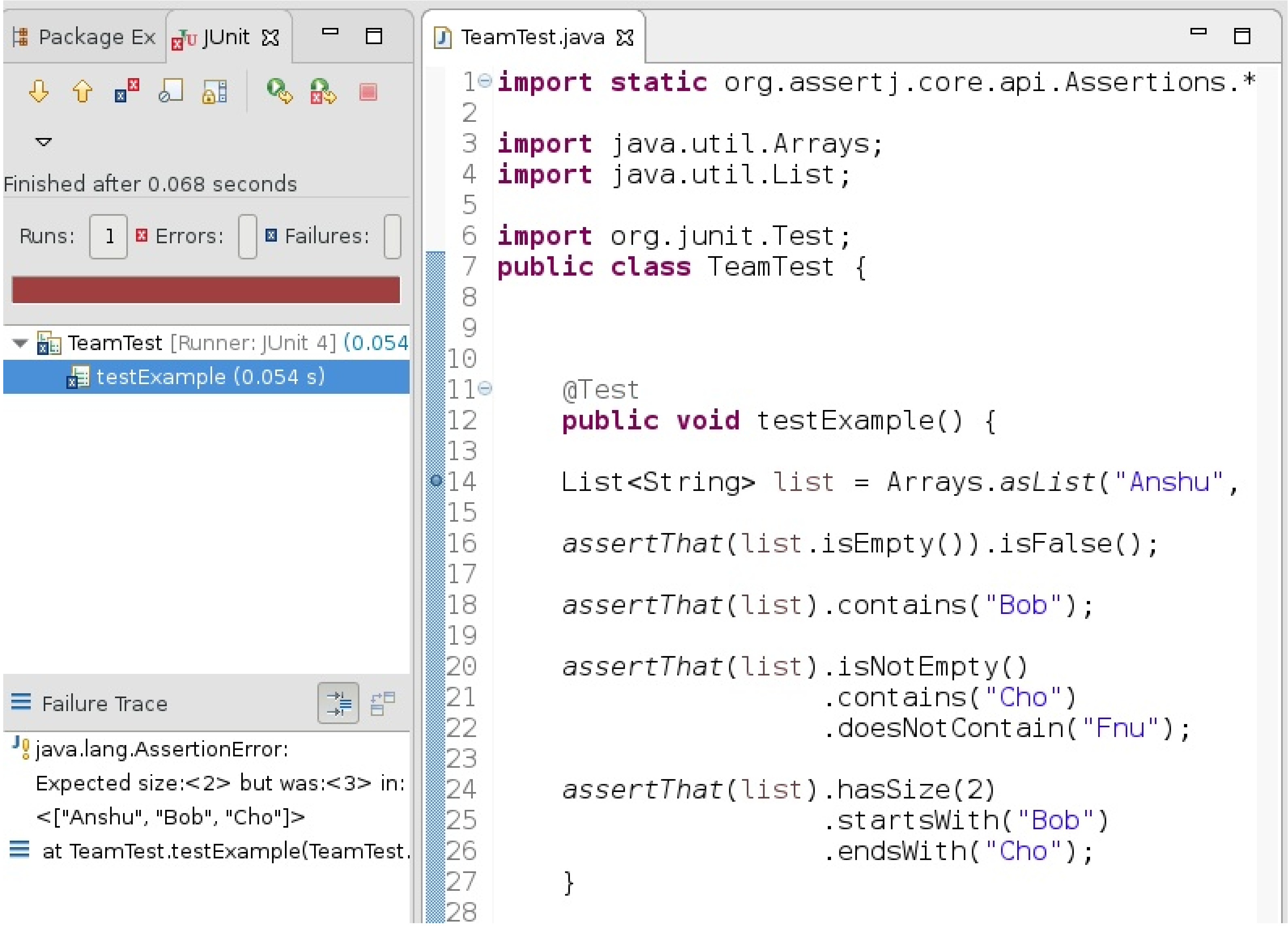
- A worked example follows to demonstrate how AssertJ is used

AssertJ and Collections



```java
import static org.assertj.core.api.Assertions.*;

import java.util.Arrays;
import java.util.List;

import org.junit.Test;
public class TeamTest {



    @Test
    public void testExample() {

    List<String> list = Arrays.asList("Anshu", "Bob", "Cho");

    assertThat(list.isEmpty()).isFalse();

    assertThat(list).contains("Bob");

    assertThat(list).isNotEmpty()
                    .contains("Cho")
                    .doesNotContain("Fnu");

    assertThat(list).hasSize(3)
                    .startsWith("Anshu")
                    .endsWith("Cho");
    }

}
```

A Failing AssertJ Assertion

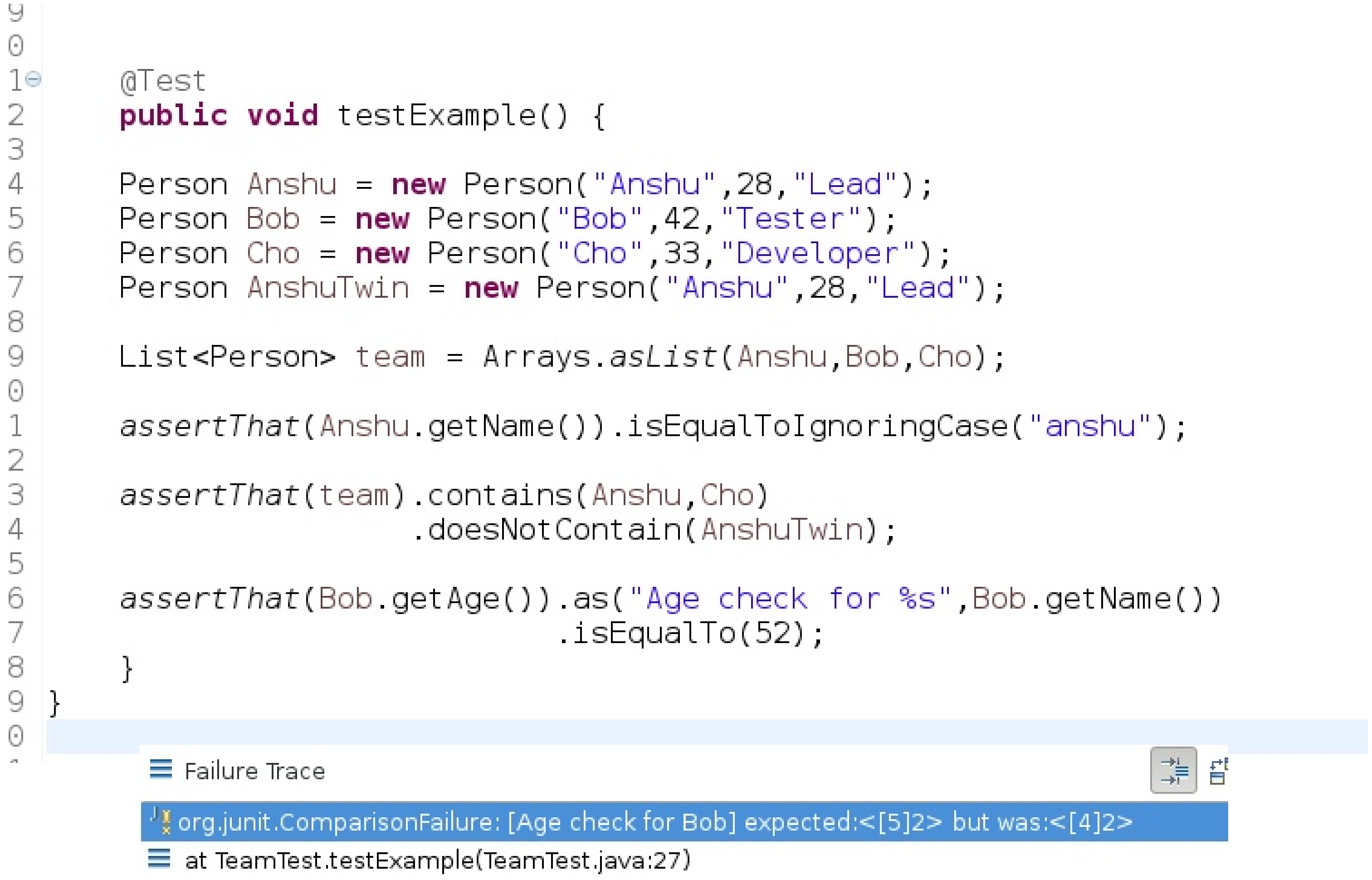


```
Finished after 0.068 seconds

Runs: 1    Errors:    Failures:

TeamTest [Runner: JUnit 4] (0.054
    testExample (0.054 s)
```

```
java.lang.AssertionError:
Expected size:<2> but was:<3> in:
<["Anshu", "Bob", "Cho"]>
at TeamTest.testExample(TeamTest.
```

```java
import static org.assertj.core.api.Assertions.*

import java.util.Arrays;
import java.util.List;

import org.junit.Test;
public class TeamTest {



    @Test
    public void testExample() {

    List<String> list = Arrays.asList("Anshu",

    assertThat(list.isEmpty()).isFalse();

    assertThat(list).contains("Bob");

    assertThat(list).isNotEmpty()
                    .contains("Cho")
                    .doesNotContain("Fnu");

    assertThat(list).hasSize(2)
                    .startsWith("Bob")
                    .endsWith("Cho");
    }

```

More AssertJ Assertion Predicates



```
    @Test
    public void testExample() {

    Person Anshu = new Person("Anshu",28,"Lead");
    Person Bob = new Person("Bob",42,"Tester");
    Person Cho = new Person("Cho",33,"Developer");
    Person AnshuTwin = new Person("Anshu",28,"Lead");

    List<Person> team = Arrays.asList(Anshu,Bob,Cho);

    assertThat(Anshu.getName()).isEqualToIgnoringCase("anshu");

    assertThat(team).contains(Anshu,Cho)
                    .doesNotContain(AnshuTwin);

    assertThat(Bob.getAge()).as("Age check for %s",Bob.getName())
                            .isEqualTo(52);
    }
}
```

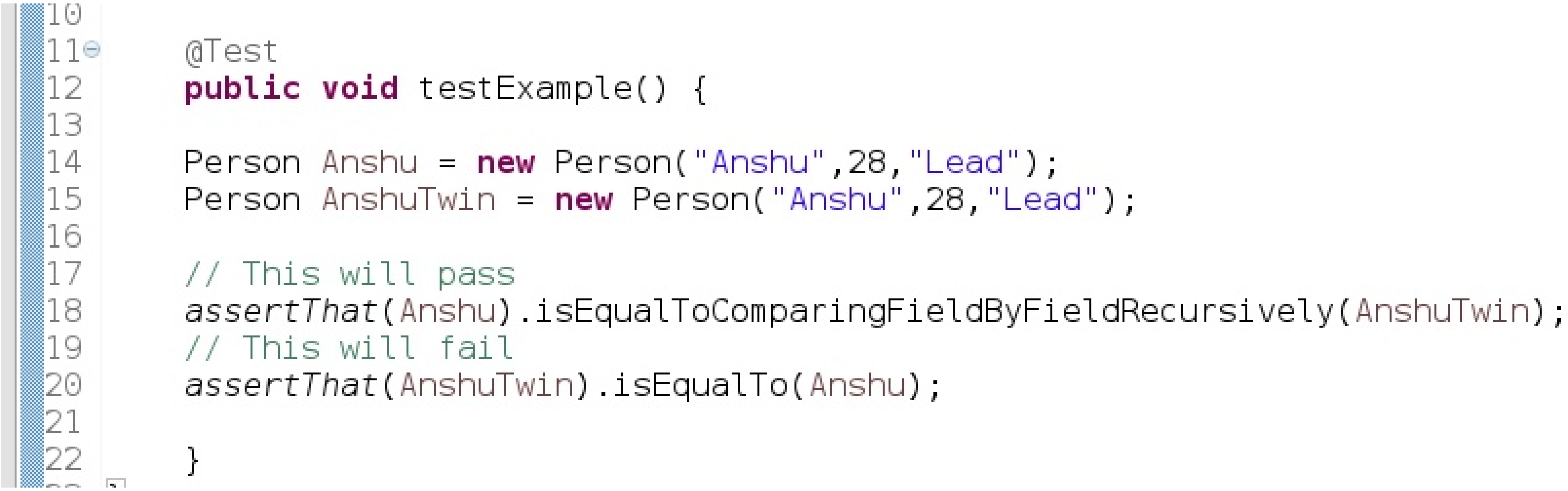
```
org.junit.ComparisonFailure: [Age check for Bob] expected:<[5]2> but was:<[4]2>
at TeamTest.testExample(TeamTest.java:27)
```

More Matchers

The example shows more matchers and how they are used. At this point the syntax should be clear.

The last line is a failing assertion to demonstrate how we can format output using the “.as()” clause to print out useful messages when an assertion fails.

Object Equality and Equivalence



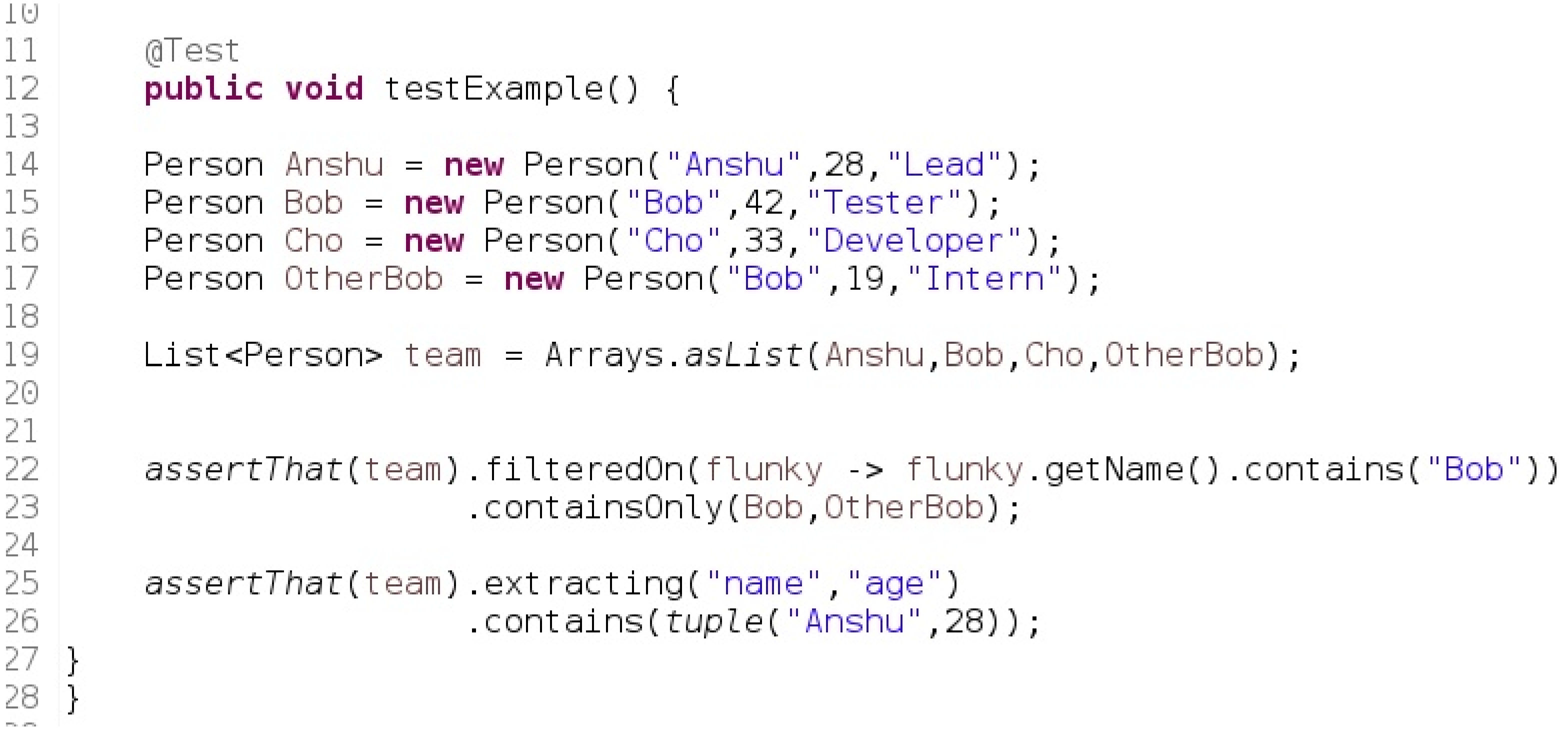
```
@Test
public void testExample() {

Person Anshu = new Person("Anshu",28,"Lead");
Person AnshuTwin = new Person("Anshu",28,"Lead");

// This will pass
assertThat(Anshu).isEqualToComparingFieldByFieldRecursively(AnshuTwin);
// This will fail
assertThat(AnshuTwin).isEqualTo(Anshu);

}
```

Equality and Equivalence

In OOP we say that two references to an object are “equal” is they refer to the same object in memory, i.e. point to the same memory location. We say that two references are equivalent if both references point to objects that are the same type and that every instance variable in one object has the same value as the corresponding instance variable in the other object.

In the example above, Anshu and AnshuTwin are different physical objects so they are not equal. However Anshu and AnshuTwin are equivalent.

Filtering and Extracting



```
    @Test
    public void testExample() {

    Person Anshu = new Person("Anshu",28,"Lead");
    Person Bob = new Person("Bob",42,"Tester");
    Person Cho = new Person("Cho",33,"Developer");
    Person OtherBob = new Person("Bob",19,"Intern");

    List<Person> team = Arrays.asList(Anshu,Bob,Cho,OtherBob);


    assertThat(team).filteredOn(flunky -> flunky.getName().contains("Bob"))
                    .containsOnly(Bob,OtherBob);

    assertThat(team).extracting("name","age")
                    .contains(tuple("Anshu",28));
}
}
```

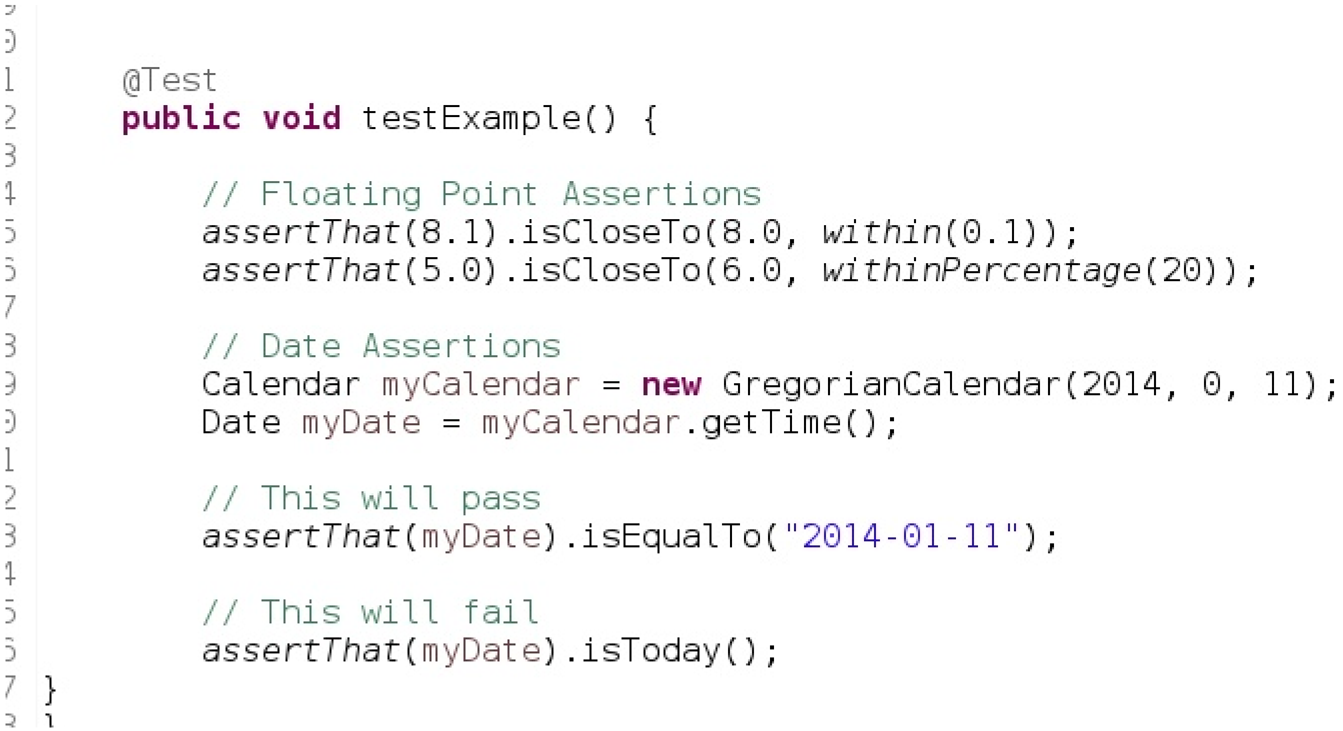
Filtering and Extracting

For complex objects or collections we are often interested in some subset of the object. Filters allow us to pull out a subset of the collection based on some value. In this example about, we are only interested in the team members named “Bob”. We could write the code to extract these manually but using an AssertJ filter is a lot less work.

Extracting is similar but instead of selecting some subset of the data, we are instead only interested in some attributes of the objects that make up the collection. IN the above example, we are only interested in the “name” and “age” properties of Person objects that make up the team.

If you are familiar with SQL, a filter is like a SELECT statement and extracting is like creating a projection of table by dropping columns.

Assertions on Data Types



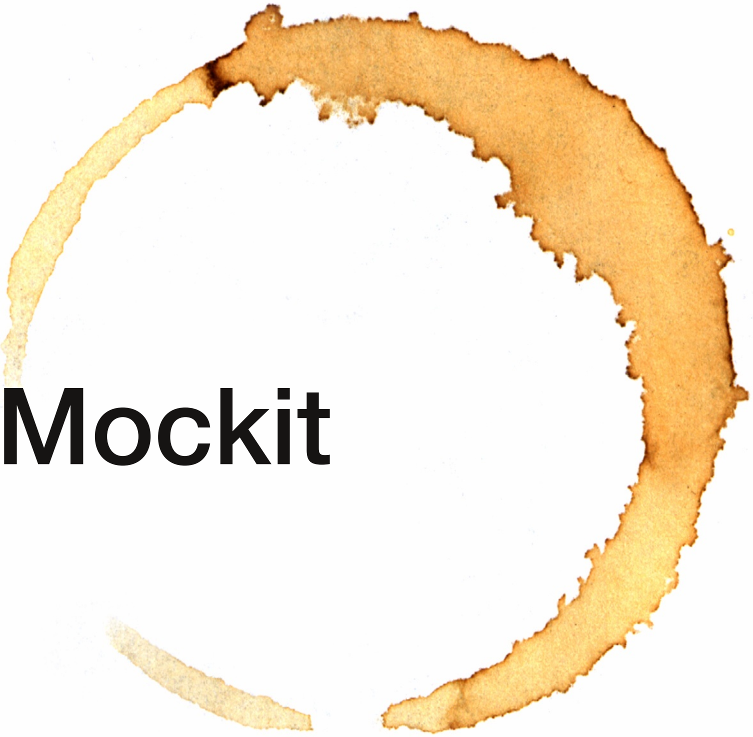
```
    @Test
    public void testExample() {

        // Floating Point Assertions
        assertThat(8.1).isCloseTo(8.0, within(0.1));
        assertThat(5.0).isCloseTo(6.0, withinPercentage(20));

        // Date Assertions
        Calendar myCalendar = new GregorianCalendar(2014, 0, 11);
        Date myDate = myCalendar.getTime();

        // This will pass
        assertThat(myDate).isEqualTo("2014-01-11");

        // This will fail
        assertThat(myDate).isToday();
}
```

Assertions on Data Types

Floating point numbers, because of precision and rounding errors, are never exactly equal but are equal within some tolerance or range of precision. The AssertJ matchers allow expressing this concept in a natural manner and syntax.

Similarly, there are specific matchers that cab be used with Java data types – in the example we are creating a date and then checking to see if the date is today's date.

Mocking with JMockit



Mocking Libraries

- We created a mock database by hand earlier it the course

- We can simplify this process and make it more powerful by using a mocking library, some popular ones are:
  - *EasyMock*
  - *Mockito*
  - *PowerMock*
  - *JMockit*

- Mocking libraries are of two types:
  - *Proxy mocks which take the place of the real object – our hand crafted mock earlier was a proxy mock*
  - *API Instrumentation: where the mock class file is loaded by the Java class loader*
  - *The second can be thought of as "hot swapping" out the original class definition with the mocked definition at the JVM level*

Mock Phases in a Test

- Within a test method, a mock goes through three stages
- First, is the recording stage where the expected values are “wired” into the mock
  - *JMockit uses the “New Expectations(){{ }};”*
- Second, the recorded values are played back as the test is executed
- Third, there is an optional verification section to ensure that the mocks performed as expected

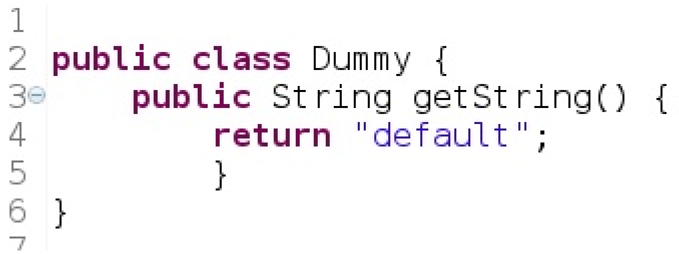
Simple Example

Simple JMockit Example

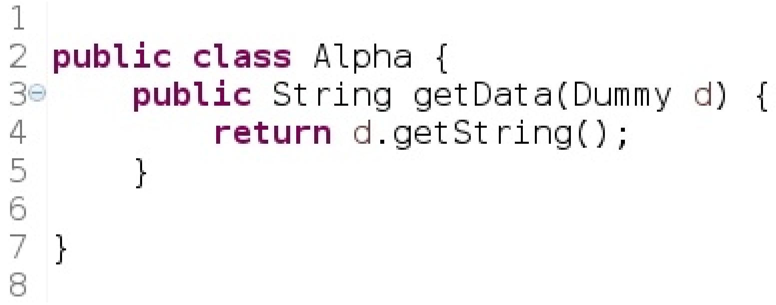
In this simple example, we are going to set up a class to be tested, called Alpha, and a class to be mocked, called Dummy.

In order to see that our mock works, the only method defined in Dummy returns the string “default.”

The class under test, Alpha, calls the method getString() on Dummy. What we want to do is use a mock for Dummy so that we can override the existing method and return whatever we want.



```

public class Dummy {
    public String getString() {
        return "default";
        }
}
```



```

public class Alpha {
    public String getData(Dummy d) {
        return d.getString();
    }

}

```

Simple Example Test Class

Simple JMockit Test Class

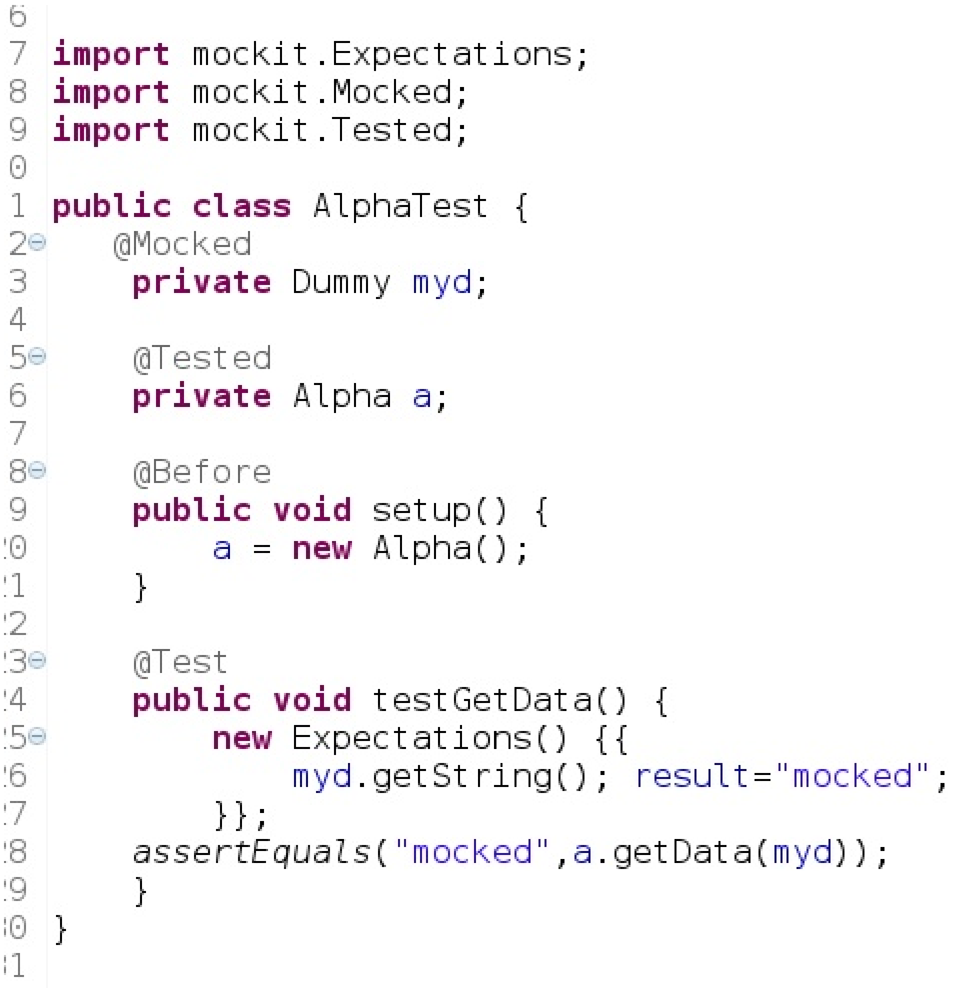
The @Test method and assertion are standard JUnit code. However we have defined private Dummy object called “myd.”

The @Mocked annotation on the declaration tells JMockit that all instances of this object are to be mocked. Since JMockit is creating mocks for us, we don’t actually create an instance of a Dummy object anywhere. In fact, if we do try and create an instance of it and call the getString() method, we will get a null back because JMockit will still use the mock object.

The @Tested annotation tells JMockit that the object “a” is the object under test.

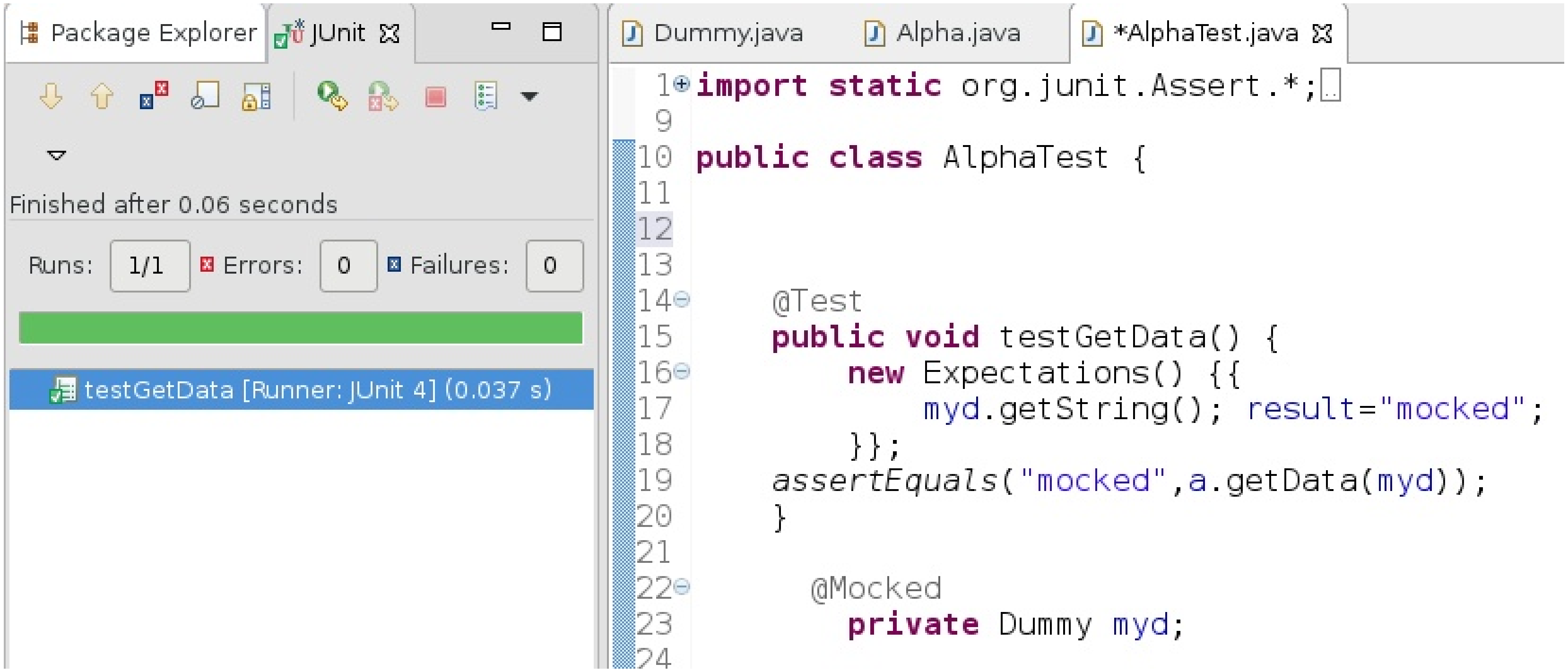
In the test method, the Expectations() clause says that when the method call myd.getString() is executed, then the result “mocked” is to be returned.

In the assertion we can see that in fact that is exactly what happened.



```
import mockit.Expectations;
import mockit.Mocked;
import mockit.Tested;

public class AlphaTest {
   @Mocked
    private Dummy myd;

    @Tested
    private Alpha a;

    @Before
    public void setup() {
        a = new Alpha();
    }

    @Test
    public void testGetData() {
        new Expectations() {{
            myd.getString(); result="mocked";
        }};
        assertEquals("mocked",a.getData(myd));
    }
}
```

Simple Example Test Class



Finished after 0.06 seconds

Runs: 1/1 Errors: 0 Failures: 0

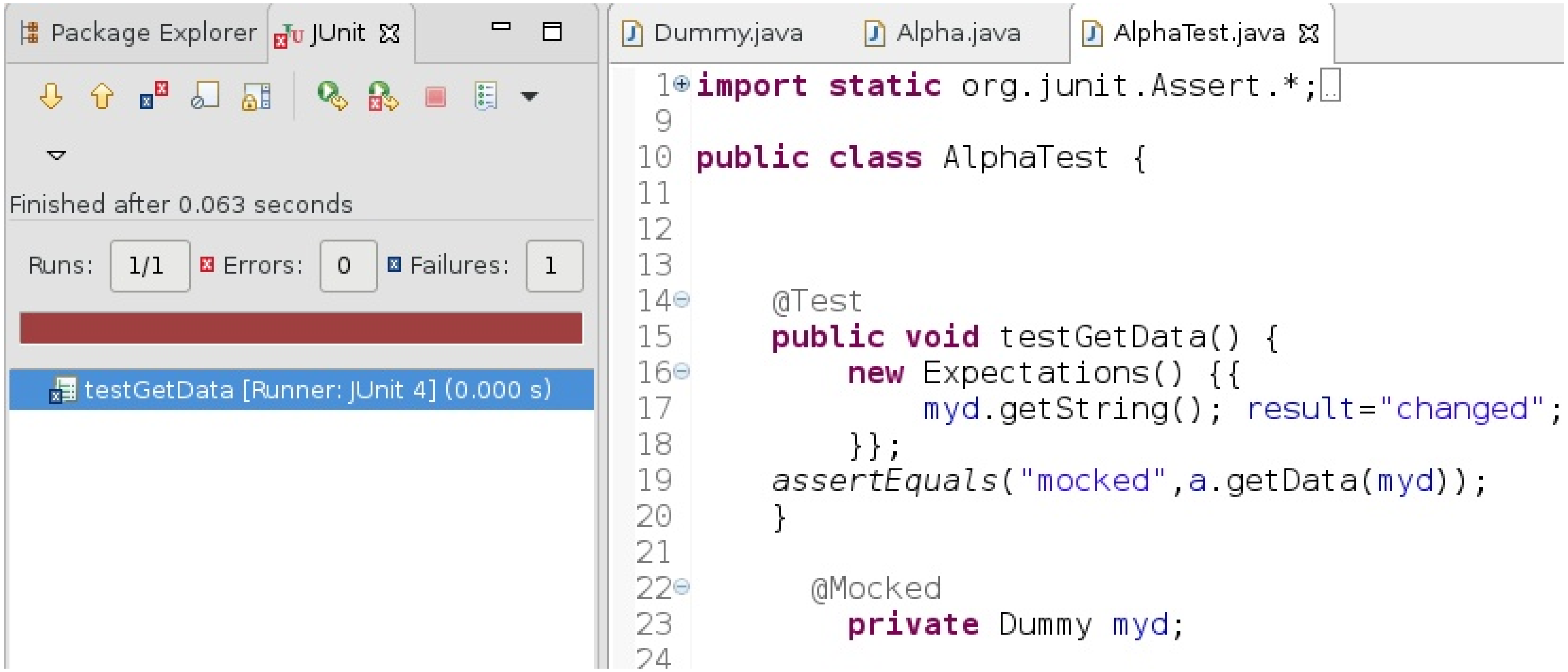
testGetData [Runner: JUnit 4] (0.037 s)

```java
import static org.junit.Assert.*;

public class AlphaTest {



    @Test
    public void testGetData() {
        new Expectations() {{
            myd.getString(); result="mocked";
        }};
    assertEquals("mocked",a.getData(myd));
    }

      @Mocked
        private Dummy myd;

```

Simple JMockit Test Result

Running the JUnit test shows the mock worked.

Simple Example Test Class



Finished after 0.063 seconds

Runs: 1/1 Errors: 0 Failures: 1

testGetData [Runner: JUnit 4] (0.000 s)

```
import static org.junit.Assert.*;

public class AlphaTest {



    @Test
    public void testGetData() {
        new Expectations() {{
            myd.getString(); result="changed";
        }};
    assertEquals("mocked",a.getData(myd));
    }

      @Mocked
        private Dummy myd;
```

Simple JMockit Test Result

Changing the value of the expectation in the recording now produces a failure when it is played back.

Injectable Mocks

- In the examples so far, all instances of Dummy are mocked

- By using the @Injectable annotation:
  - *Only a specific instance is mocked*
  - *All other instances are not mocked*

- The mocked instance is injected as a parameter to a test method
  - *The @Mocked annotation identifies the parameter as a mock*

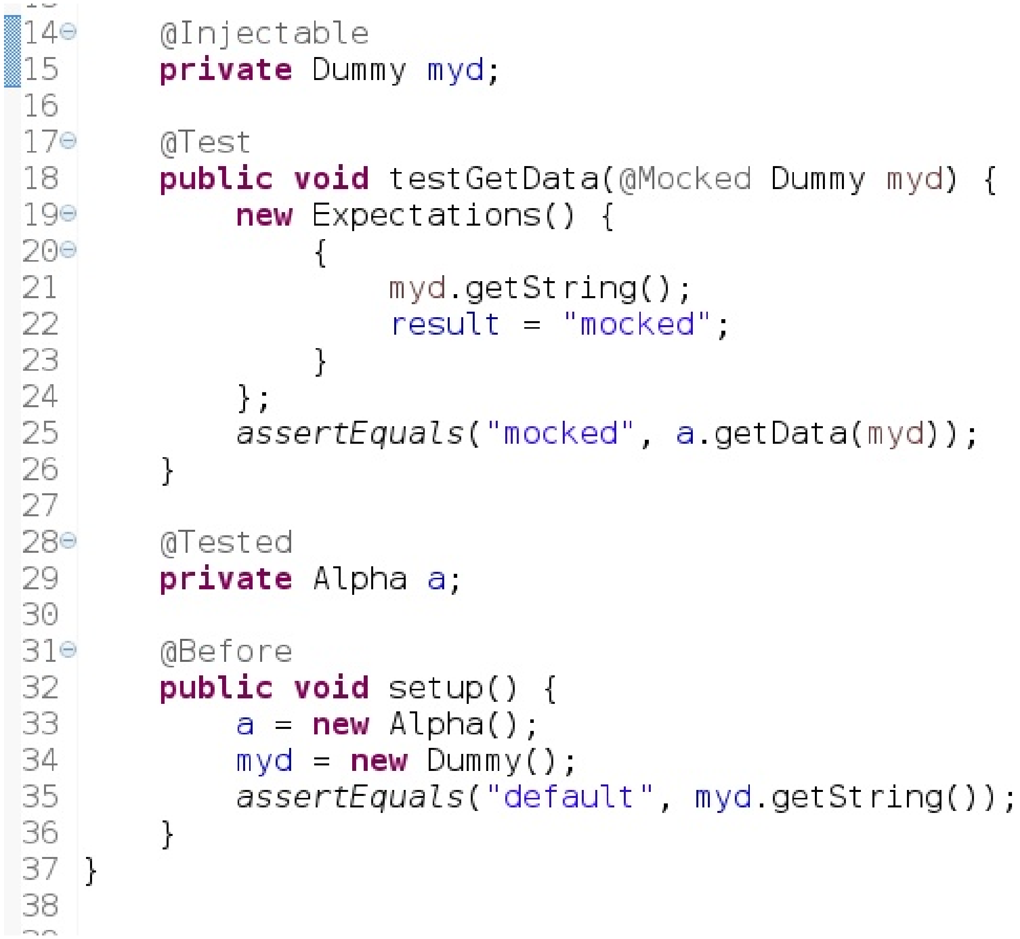
Injectable Mock Example

Injectable Mocks

The examples shows the use of the @Injectable and @Mocked annotations to inject the mock into the test method.

In the @Before method, an instance of Dummy is created, as as can be seen by the assertion, is a not a mock.

However, once the @Test method executes, the mock for Dummy is used. The mock is used only within the scope of the test method.

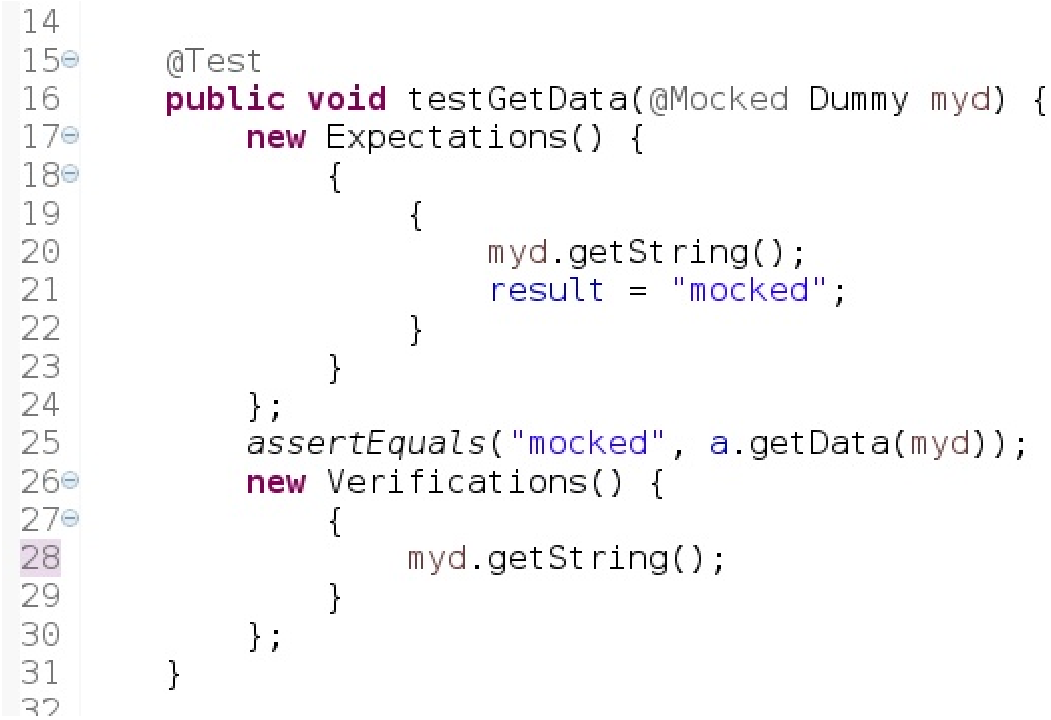


```
    @Injectable
    private Dummy myd;

    @Test
    public void testGetData(@Mocked Dummy myd) {
        new Expectations() {
            {
                myd.getString();
                result = "mocked";
            }
        };
        assertEquals("mocked", a.getData(myd));
    }

    @Tested
    private Alpha a;

    @Before
    public void setup() {
        a = new Alpha();
        myd = new Dummy();
        assertEquals("default", myd.getString());
    }
}
```

Verifications

- A verification block can be added to ensure that the mock worked
- A verification is not a test, it is ensures the test ran properly
- If a verification fails, then the test fails
  - *JMockit assumes that if a mock did not run properly, then the test results are suspect.*
  - *Since the test results cannot be assured, the test fails*

A Simple Verification



```
    @Test
    public void testGetData(@Mocked Dummy myd) {
        new Expectations() {
            {
                {
                    myd.getString();
                    result = "mocked";
                }
            }
        };
        assertEquals("mocked", a.getData(myd));
        new Verifications() {
            {
                myd.getString();
            }
        };
    }
```

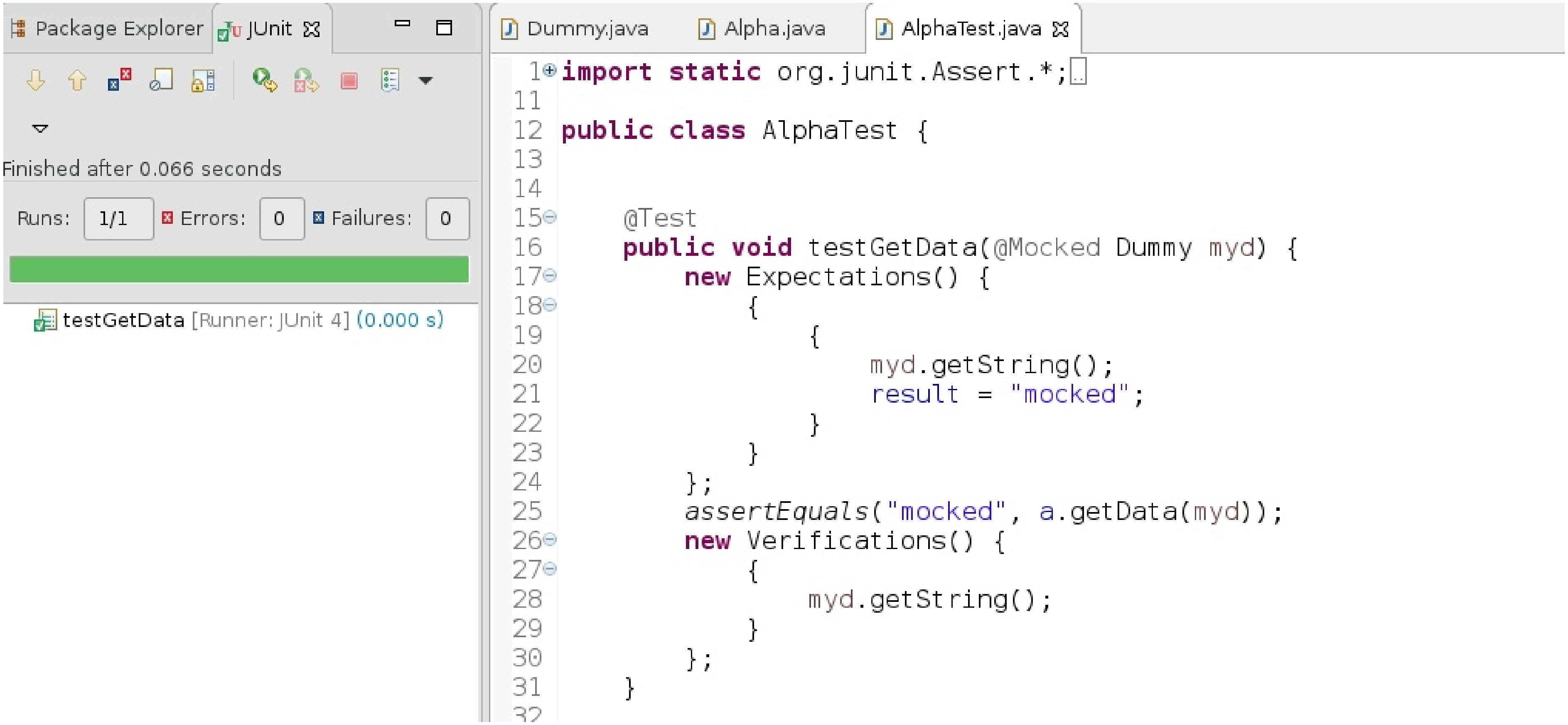
Simple Verification

In the verification block, we have asserted that the mocked method call must occur. If we do not make at least one method call to the mocked method, then the verification will not be true and the test will fail.

If we run this test method in the current form, since there is a method call in the method which uses the mock, the verification passes.

This is shown in the next slide.

A Simple Verification



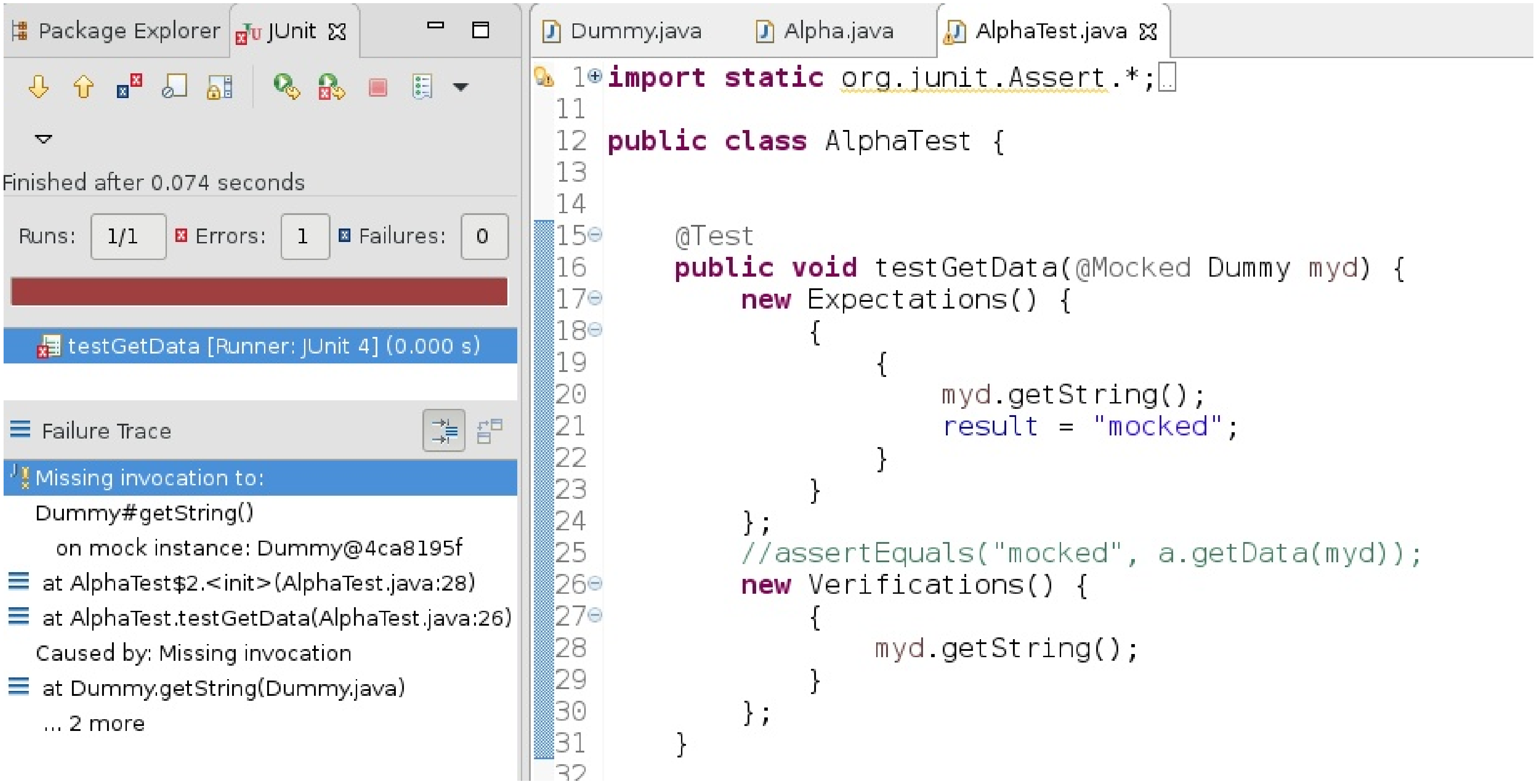
Finished after 0.066 seconds

Runs: 1/1 Errors: 0 Failures: 0

testGetData [Runner: JUnit 4] (0.000 s)

```
import static org.junit.Assert.*;

public class AlphaTest {


    @Test
    public void testGetData(@Mocked Dummy myd) {
        new Expectations() {
            {
                {
                    myd.getString();
                    result = "mocked";
                }
            }
        };
        assertEquals("mocked", a.getData(myd));
        new Verifications() {
            {
                myd.getString();
            }
        };
    }
```

A Simple Verification Failure



Finished after 0.074 seconds

Runs: 1/1 ☒ Errors: 1 ☒ Failures: 0

testGetData [Runner: JUnit 4] (0.000 s)

Failure Trace

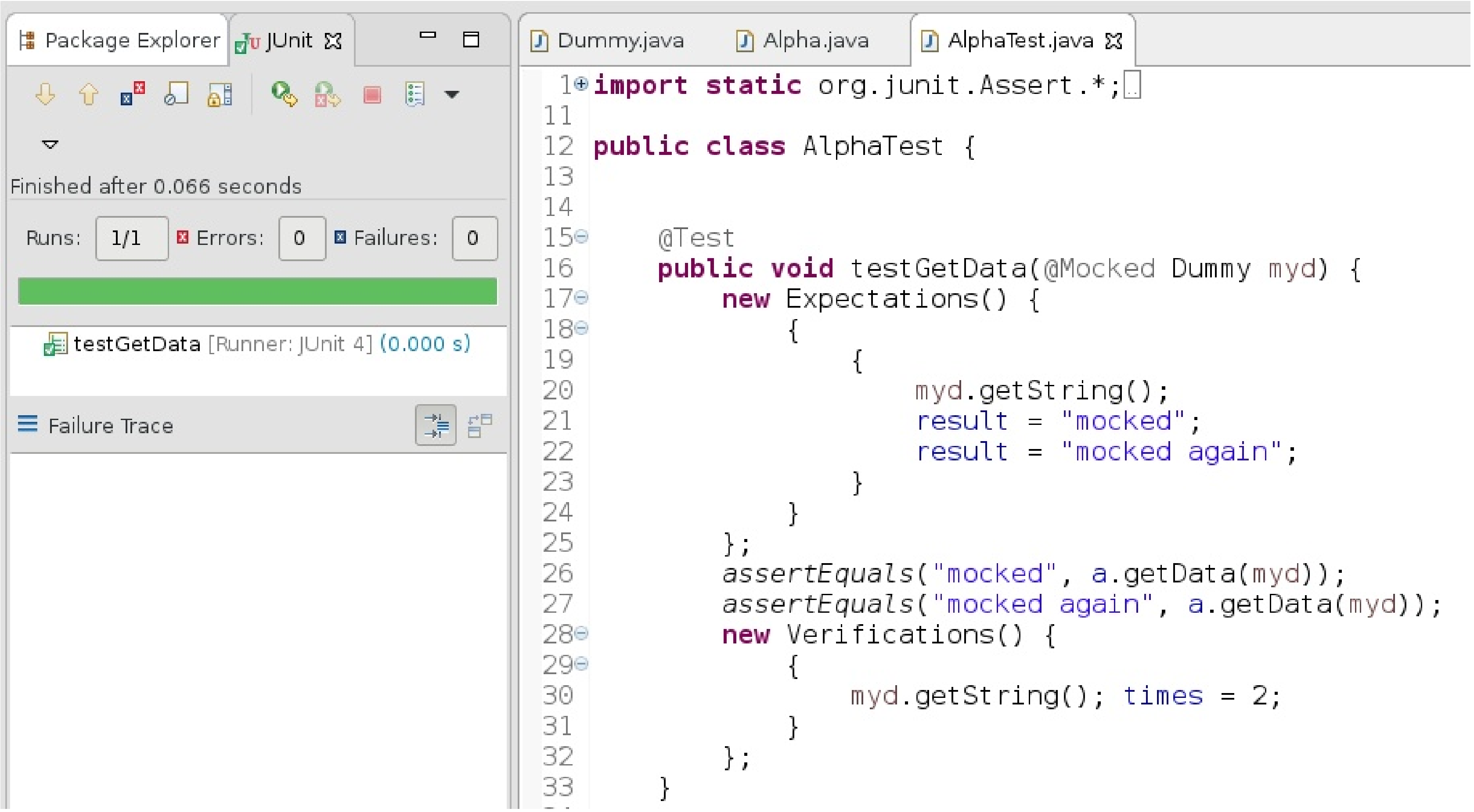
```
Missing invocation to:
Dummy#getString()
   on mock instance: Dummy@4ca8195f
at AlphaTest$2.<init>(AlphaTest.java:28)
at AlphaTest.testGetData(AlphaTest.java:26)
Caused by: Missing invocation
at Dummy.getString(Dummy.java)
 ... 2 more
```

```java
import static org.junit.Assert.*;

public class AlphaTest {


    @Test
    public void testGetData(@Mocked Dummy myd) {
        new Expectations() {
            {
                {
                    myd.getString();
                    result = "mocked";
                }
            }
        };
        //assertEquals("mocked", a.getData(myd));
        new Verifications() {
            {
                myd.getString();
            }
        };
    }
```

Simple Verification Failure

Commenting out the method call produces a verification failure.

Multiple Calls and Verifications



Finished after 0.066 seconds

Runs: 1/1 Errors: 0 Failures: 0

testGetData [Runner: JUnit 4] (0.000 s)

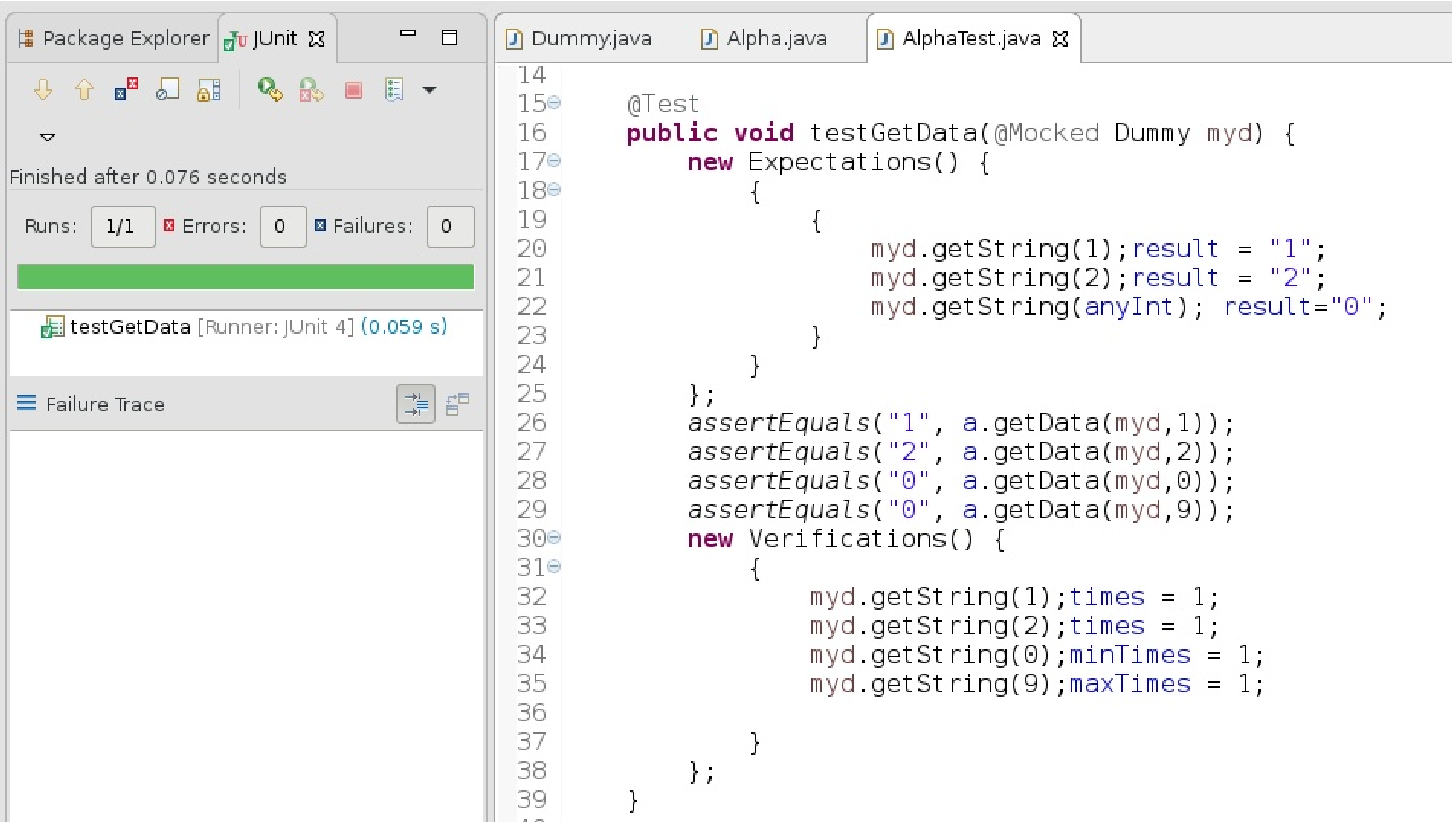
```java
import static org.junit.Assert.*;

public class AlphaTest {


    @Test
    public void testGetData(@Mocked Dummy myd) {
        new Expectations() {
            {
                {
                    myd.getString();
                    result = "mocked";
                    result = "mocked again";
                }
            }
        };
        assertEquals("mocked", a.getData(myd));
        assertEquals("mocked again", a.getData(myd));
        new Verifications() {
            {
                myd.getString(); times = 2;
            }
        };
    }
```

Multiple Method Calls

We can specific a different expected return value each time a method is called by supplying a list of results. If there are more calls than values in the list, the last value is reused.

We can also specify in the verification block the exact number of times a method should be called using the times option. We can also specify using the minTimes and maxTimes options.

Matching Arguments



Finished after 0.076 seconds

Runs: 1/1 Errors: 0 Failures: 0

testGetData [Runner: JUnit 4] (0.059 s)

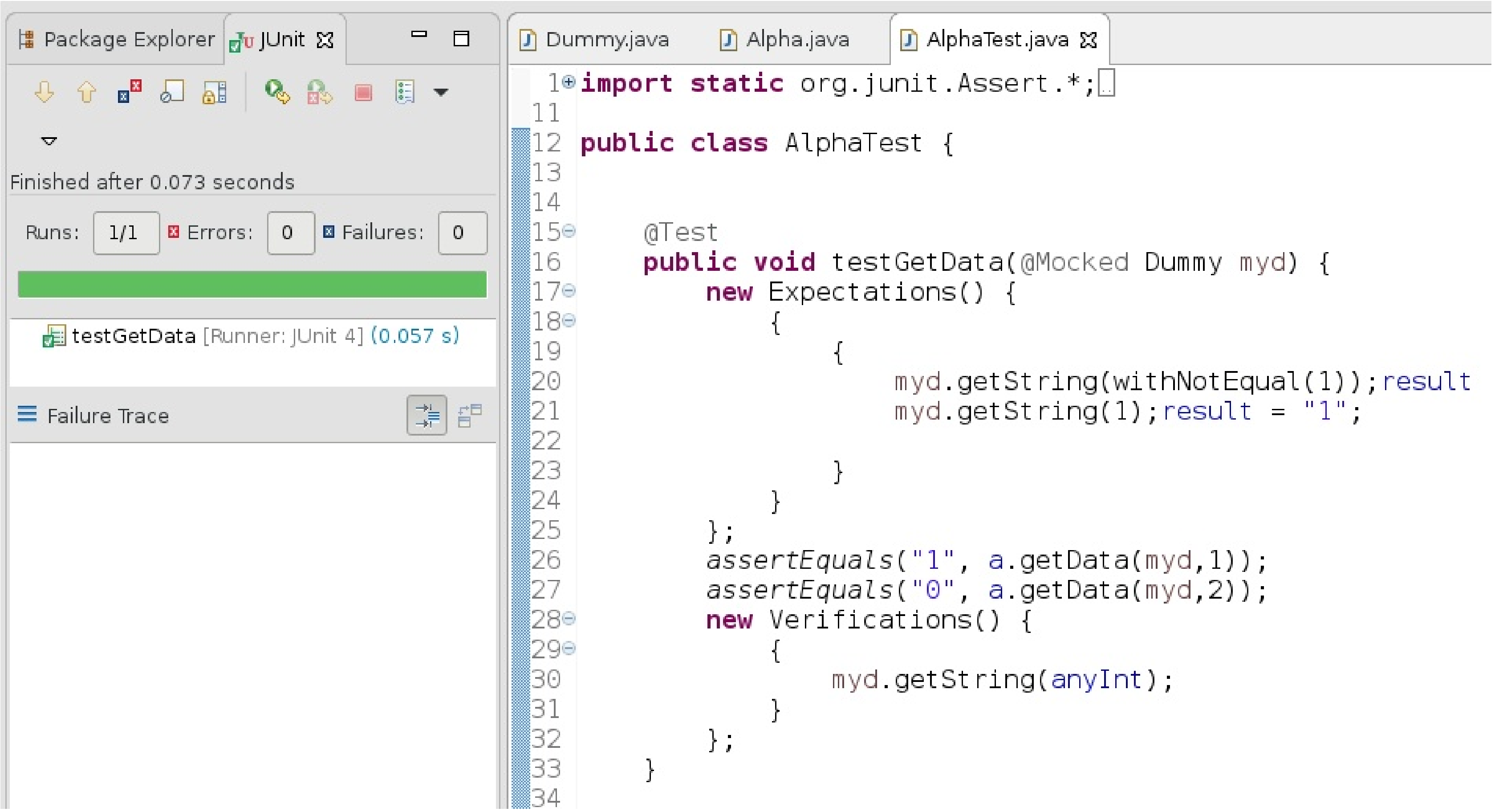
```java

    @Test
    public void testGetData(@Mocked Dummy myd) {
        new Expectations() {
            {
                {
                    myd.getString(1);result = "1";
                    myd.getString(2);result = "2";
                    myd.getString(anyInt); result="0";
                }
            }
        };
        assertEquals("1", a.getData(myd,1));
        assertEquals("2", a.getData(myd,2));
        assertEquals("0", a.getData(myd,0));
        assertEquals("0", a.getData(myd,9));
        new Verifications() {
            {
                myd.getString(1);times = 1;
                myd.getString(2);times = 1;
                myd.getString(0);minTimes = 1;
                myd.getString(9);maxTimes = 1;

            }
        };
    }
```

Matching Arguments

We can adjust the expected value of the mock method based on the values of the arguments. The example has been re-written so now the getString() and getData() take an integer parameter.

We can specify the return value for the mocked up method based on the parameter value. Notice the use of anyInt as a placeholder that matches any integer parameter.

Matching Argument Patterns



```
Finished after 0.073 seconds
Runs: 1/1   Errors: 0   Failures: 0
testGetData [Runner: JUnit 4] (0.057 s)
```

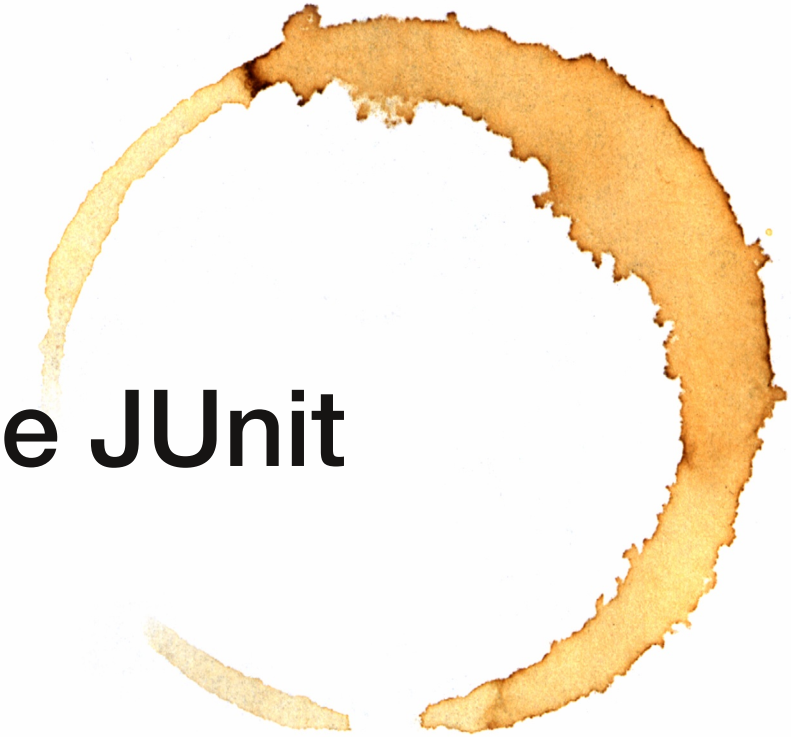
```java
import static org.junit.Assert.*;

public class AlphaTest {


    @Test
    public void testGetData(@Mocked Dummy myd) {
        new Expectations() {
            {
                {
                    myd.getString(withNotEqual(1));result
                    myd.getString(1);result = "1";

                }
            }
        };
        assertEquals("1", a.getData(myd,1));
        assertEquals("0", a.getData(myd,2));
        new Verifications() {
            {
                myd.getString(anyInt);
            }
        };
    }

```

Matching Argument Patterns

We can use the “with” operators to match patterns of arguments or to filter arguments by some criteria. In the example above, the “withNotEqual()” matcher is used to return “0” for any argument that is not the integer 1.

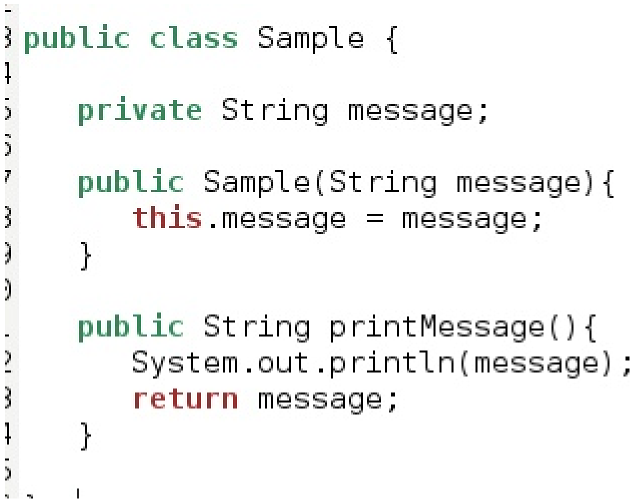
Notice the use of the anyInt matcher in the verification block. Not specifying a “times” clause causes the verification block to just ensure that the mock method was called at least once with some integer as an argument.

Command Line JUnit

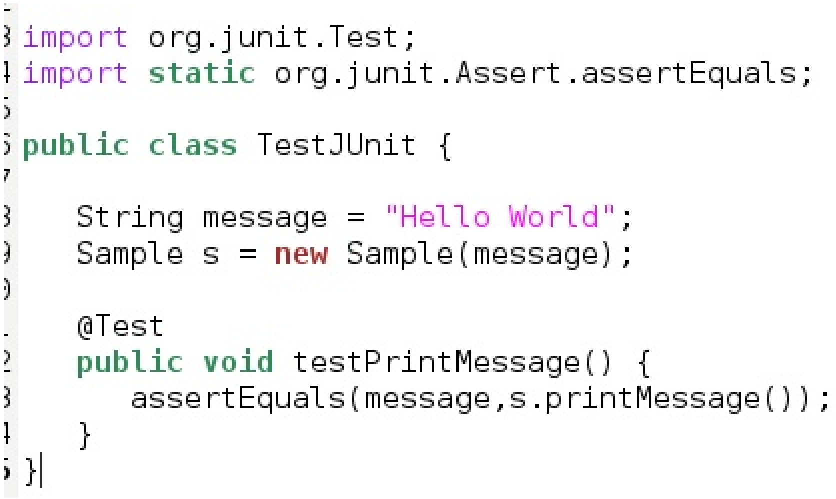


Running JUnit at the Command Line

- So far all of the JUnit tests have been run through eclipse
- We can run JUnit tests from other environments including the command line
- In the following example, we run a simple test at the command line
  - *The class under test is trivial and shown below with the test class*

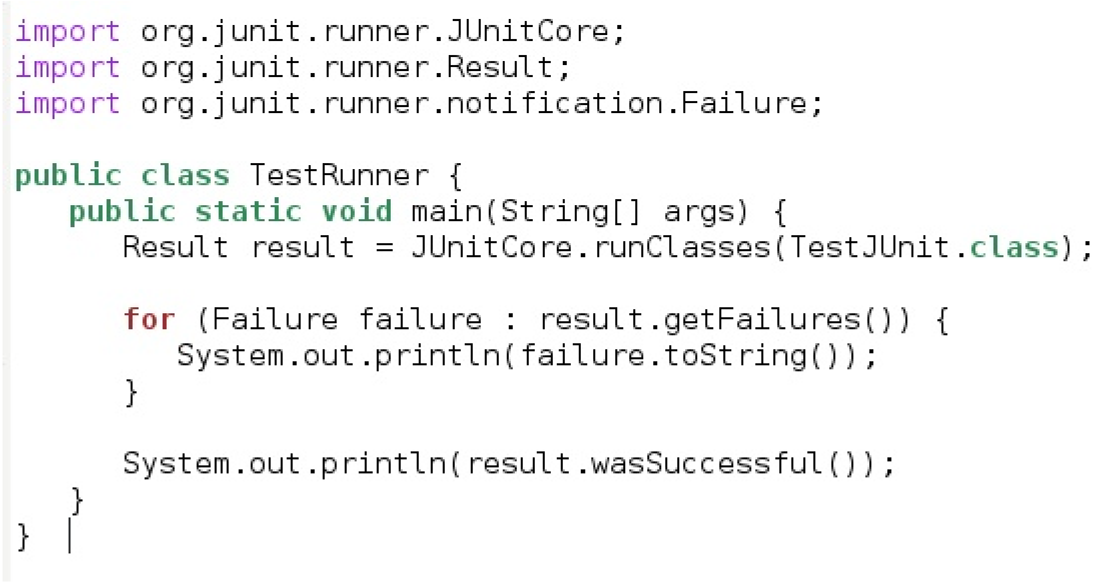


```
public class Sample {

    private String message;

    public Sample(String message){
        this.message = message;
    }

    public String printMessage(){
        System.out.println(message);
        return message;
    }

```

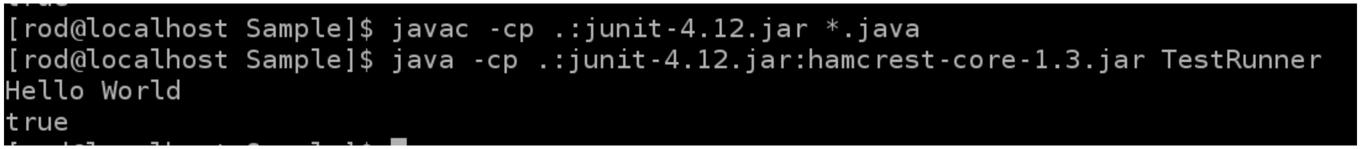


```
import org.junit.Test;
import static org.junit.Assert.assertEquals;

public class TestJUnit {

    String message = "Hello World";
    Sample s = new Sample(message);

    @Test
    public void testPrintMessage() {
        assertEquals(message,s.printMessage());
    }
}
```

The Runner Class



```java
import org.junit.runner.JUnitCore;
import org.junit.runner.Result;
import org.junit.runner.notification.Failure;

public class TestRunner {
   public static void main(String[] args) {
      Result result = JUnitCore.runClasses(TestJUnit.class);

      for (Failure failure : result.getFailures()) {
         System.out.println(failure.toString());
      }

      System.out.println(result.wasSuccessful());
   }
}
```



```
[rod@localhost Sample]$ javac -cp .:junit-4.12.jar *.java
[rod@localhost Sample]$ java -cp .:junit-4.12.jar:hamcrest-core-1.3.jar TestRunner
Hello World
true
```

Runner Class

The runner class is a Java application where the main method uses the JUnit runClasses() method to run the test class from the previous slide. This is exactly what Eclipse does in the background when we run a JUnit test case. The method returns a Result object which encapsulates the results of running the test. Normally the results are then sent through some form of results formatter that prepares the results to be used or read. In this case the formatter is just a simple command line output statement.

Experimenting with this at the command line is left as an exercise.

End of Module 6