infoSupport
Solid Innovator

Advanced use of JUnit.org

Content

- Test Suites
- Test categories
- Parameterized tests
- JUnit rules

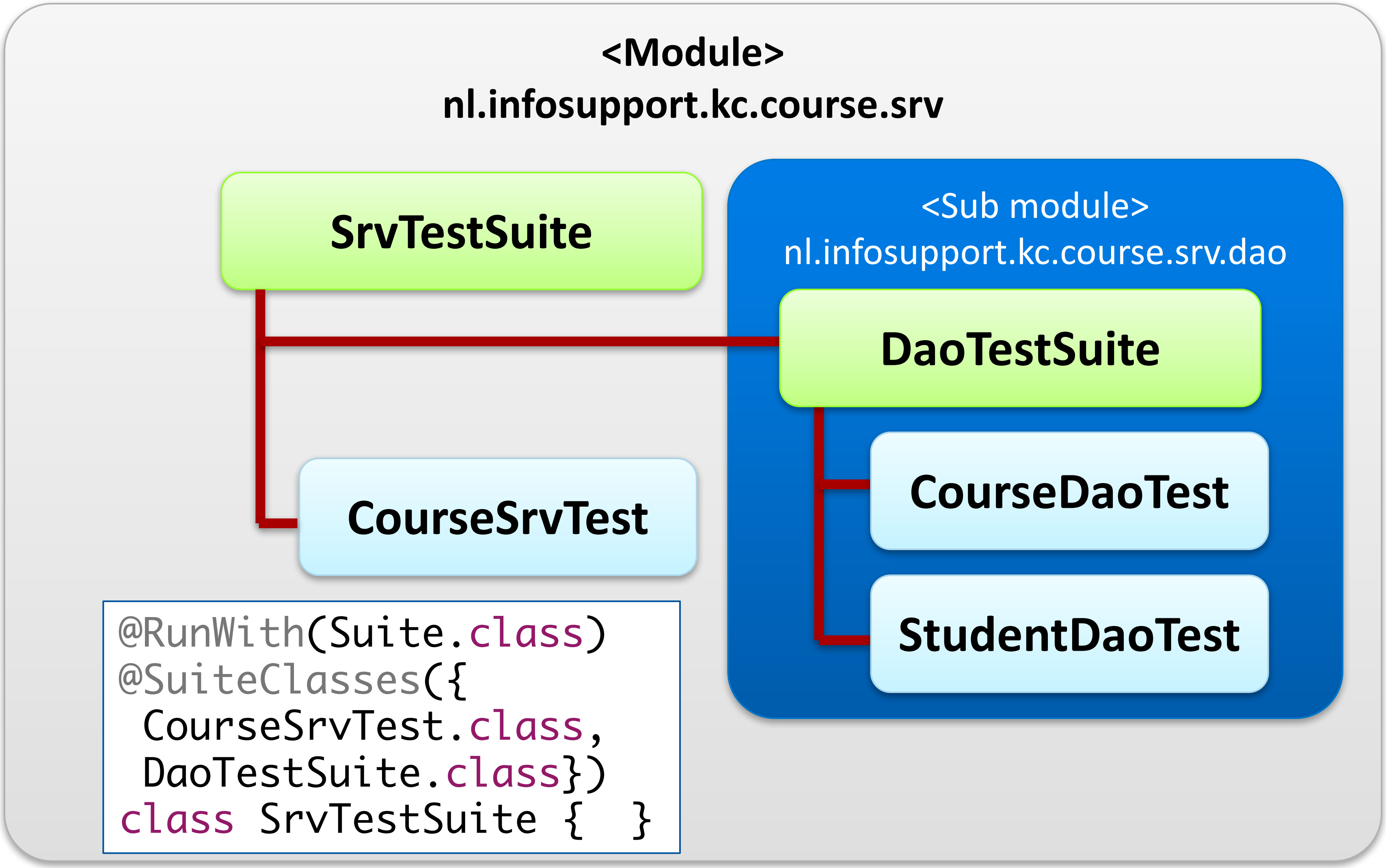
Aggregating tests

- To run all- or a group of tests, it’s helpful to group the tests in a single Test suite
- Rule: create for every test package a test suite
- JUnit creating a suite: start with an empty class and use annotations

```
@RunWith(Suite.class)
@SuiteClasses({
 BookDaoTest.class,
 PersonDaoTest.class,
 CourseDaoTest.class,
 CarDaoTest.class})
public class DaoTestSuite {  }
```

- Run this suite with JUnit

Organize suites;suites within suites



```
@RunWith(Suite.class)
@SuiteClasses({
 CourseSrvTest.class,
 DaoTestSuite.class})
class SrvTestSuite {  }
```

Test categories

- There are several types of unit tests
- Sometimes, you want to run all the tests
- Sometimes, you only want to run the performance tests
- JUnit supports categorizing unit tests

Categorize your tests

- Create an empty interface

```
public interface PerformanceTests {}
```

- Add an anotation above the test method

```
@Category(PerformanceTests.class)
@Test
public void myExampleTest() {
  System.out.println("I’m a performance test");
}
```

- Or above a test class

```
@Category(PerformanceTests.class)
public class MyExampleTestClazz {
```

Categorize your tests

- Or create a specific test suite for unit tests

```
@RunWith(Categories.class)
@IncludeCategory(PerformanceTests.class)
@SuiteClasses({ SrvTestSuite.class })
public class PerformanceTestSuite {

}
```

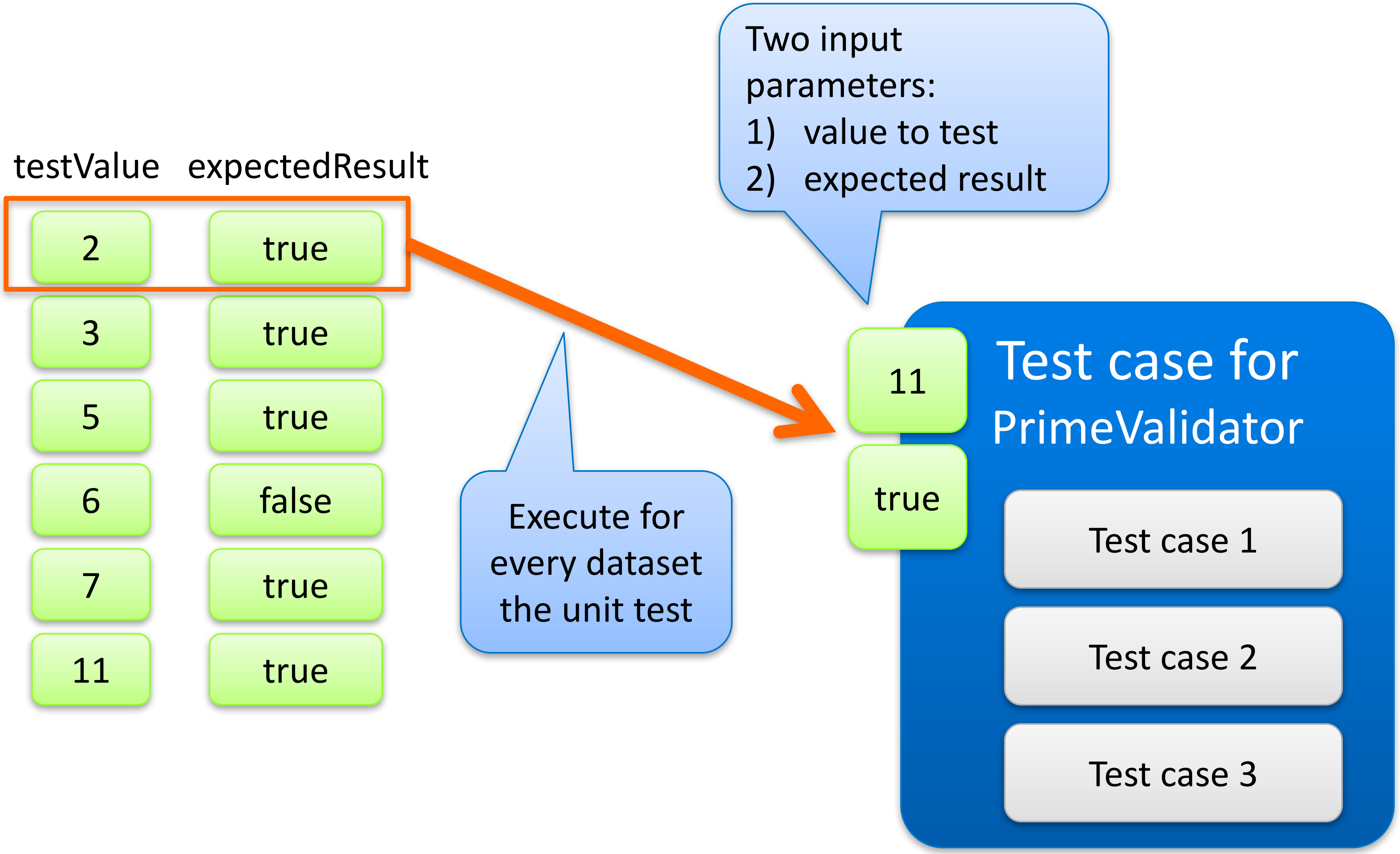
- Only the tests are executed annotated with

```
@Category(PerformanceTests.class)
```

Testing with datasets

- Assume you have to validate if a number is a prime number
- How to create a unit test case with several numbers as test value?
  - Multiple test cases?
- JUnit comes to the rescue with Parameterized test cases

Parameterized test cases



Parameterized test cases in JUnit

```
@RunWith(Parameterized.class)
public class PrimeNumberValidatorTest {

 private Integer number;
 private Boolean expect;

 public PrimeNumberValidatorTest(
     Integer number,
     Boolean expect) {
  this.number = number;
  this.expect = expect;
 }

 @Test
 public void testPrimeNumberValidator() {
  assertThat(PrimeNumberValidator.isPrime(number), is(expect));
 }

}
```

Specific RunWith annotation

Test constructor

Parameterized test cases in JUnit

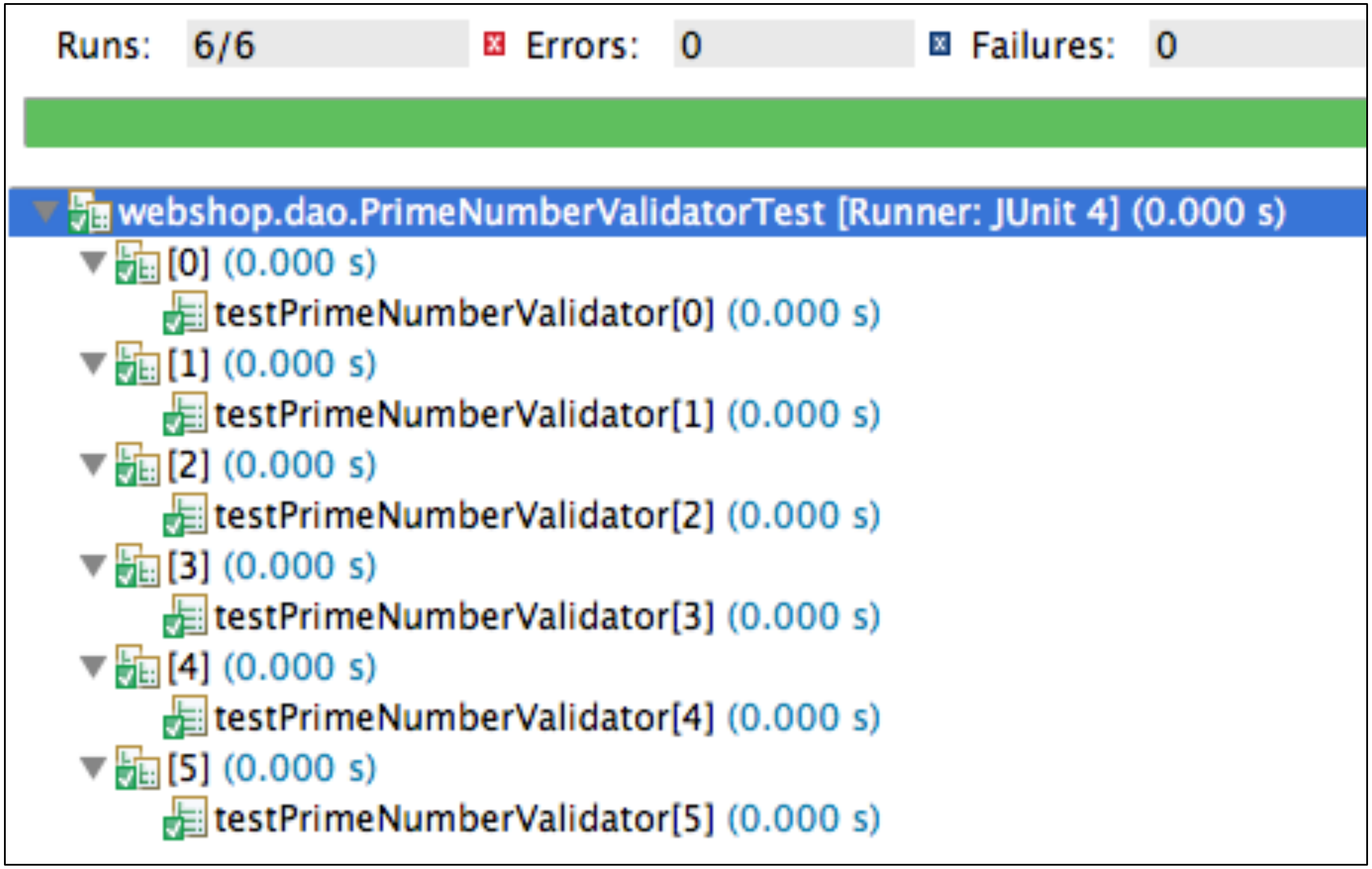
```
@RunWith(Parameterized.class)
public class PrimeNumberValidatorTest {

  @Parameterized.Parameters
  public static Collection<?> primeNumbers() {
    return Arrays.asList(new Object[][] {
        { 2, true },
        { 3, true },
        { 5, true },
        { 6, false },
        { 19, true },
        { 22, false }
    });
  }
```

Define a dataset in a static method

Values for constructor

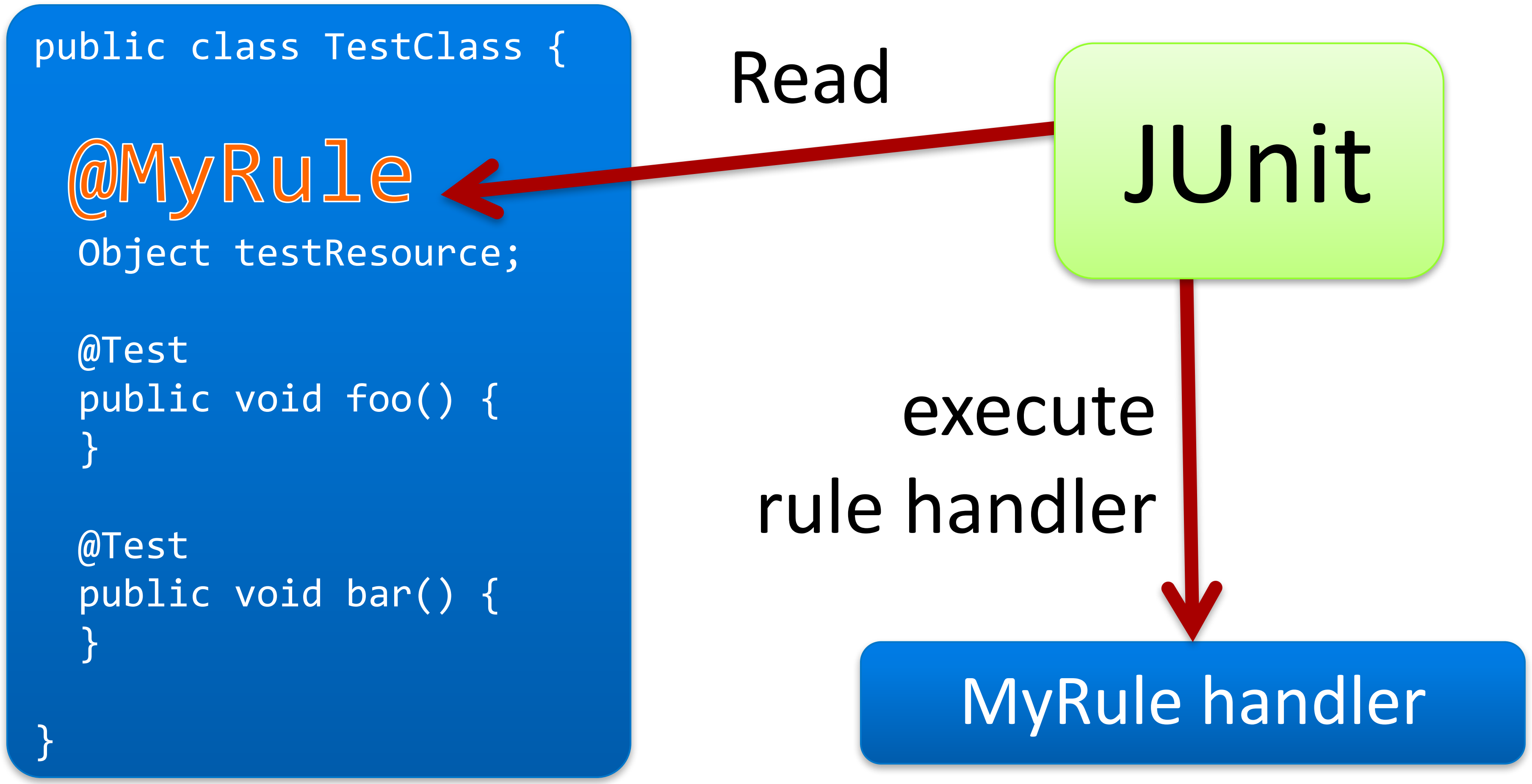
```
  public PrimeNumberVal…Test(Integer number, Boolean expect) {
```

Parameterized test cases in JUnit



```
Runs: 6/6    Errors: 0    Failures: 0

▼ webshop.dao.PrimeNumberValidatorTest [Runner: JUnit 4] (0.000 s)
  ▼ [0] (0.000 s)
      testPrimeNumberValidator[0] (0.000 s)
  ▼ [1] (0.000 s)
      testPrimeNumberValidator[1] (0.000 s)
  ▼ [2] (0.000 s)
      testPrimeNumberValidator[2] (0.000 s)
  ▼ [3] (0.000 s)
      testPrimeNumberValidator[3] (0.000 s)
  ▼ [4] (0.000 s)
      testPrimeNumberValidator[4] (0.000 s)
  ▼ [5] (0.000 s)
      testPrimeNumberValidator[5] (0.000 s)
```

Unit test @Rules



```
public class TestClass {

  @MyRule
   Object testResource;

   @Test
   public void foo() {
   }

   @Test
   public void bar() {
   }

}
```

Predefined rules

- JUnit provides a lot of predefined rules
- For example:
  - Tempory file rules
  - Max duration rules
  - Exception handling rules
  - Resource rules

Tempory files and directories

- If you want to create a tempory file or directory during a test

```
import org.junit.Rule;
import org.junit.Test;
import org.junit.rules.TemporaryFolder;

public class BookServiceTest {

  @Rule
  public TemporaryFolder tempFolder = new TemporaryFolder();

  @Test
  public void serviceCanBooksReadFromFile() {
    File file = tempFolder.newFile("books.txt");
    File tempDir = tempFolder.newFolder("tempDir");
    ...
```

Life cycle resources

- In a lot of test cases,
  - before running a test you want to open a connection or file or socket, in general a resource
  - and after running a test, you want to close it
- Without rules, you have to implement an @Before and @After method in every test class
- This can be done smarter… use Rules!
- An example follows

Life cycle of test resources

- In this example we use a server object

```
public class Server {
  public void connect() {
      System.out.println("Connect");
  }
  public void disconnect() {
      System.out.println("Disconnect");
  }
}
```

- In every test class we want to connect to and disconnect from this server

Steps to enable using resources

- First: Create a class named TestResources
- Second: Add to this class a
  - public static class ServerResource

```
public class TestResources {
  ...
  public static class ServerResource extends ExternalResource {
    private Server server = new Server();

    @Override protected void before() throws Throwable {
      server.connect();
    }
    @Override protected void after() throws Throwable {
      server.disconnect();
    }
  }
}
```

Using the resource in a test

- The defined resource can now be used in every test without @Before or @After

```
public class MyTest {
  @Rule public ServerResource resource = new ServerResource();

  @Test
  public void serviceCanBooksReadFromFile() {
    System.out.println("run test");
  }
}
```

- Output:

```
connect
run test
disconnect
```

Interacting with the Test life cycle

- TestWatcher to watch the test cycle

```
@Rule public TestWatcher watcher = new TestWatcher() {

  @Override protected void failed(
    Throwable e, Description description) {
    Toolkit.getDefaultToolkit().beep();
  }
};
```

- TestVerifier to verify a test result

```
@Rule public Verifier collector = new Verifier() {

  @Override protected void verify() {
    System.out.println("Verify");
  }
};
```

Example of a custom Log rule

```
@Rule public MyLogRule logRule = new MyLogRule();

 public static class MyLogRule implements TestRule {

  public Statement apply(final Statement base,
                                              final Description description) {
     return new Statement() {

       public void evaluate() throws Throwable {
         System.out.println("Log: " + description.getMethodName());
         base.evaluate();
       }
     };
   }
 }

 @Test
 public void methodNameTest() {
   System.out.println("Normal test method");
 }
```

Output:

```
Log: methodNameTest
Normal test method
```

Questions