Course Application Design

JUnit testing

Michiel Noback
Institute for Life Sciences and Technology
Hanze University of Applied Sciences

Contents

- In this part, we'll address a topic that is subject to heated debate in development land: testing
- Several major Java testing frameworks exist, but we will use JUnit

Unit testing

- Unit testing is, according to Wikipedia:
- *... a software testing method by which individual units of source code, sets of one or more computer program modules together with associated control data, usage procedures, and operating procedures are tested to determine if they are fit for use*

Part one

Why testing

To test or not to test

- **Test Driven Development** (**TDD**) is today’s standard in software development
- The approach we’ll take is hand-in-hand development of test and production code
- If you introduce new features a solid test suite also protects you against regression in existing code

Three reasons for unit testing (1)

- For any function and given a set of inputs, we can determine if the function is returning the proper values and will gracefully handle failures during the course of execution should invalid input be provided.

http://code.tutsplus.com/articles/the-beginners-guide-to-unit-testing-what-is-unit-testing--wp-25728

Three reasons for unit testing (2)

- You'll be writing code that is easy to test: you are more likely to have a higher number of smaller, more focused functions that provide a single operation rather than large functions performing a number of different operations

http://code.tutsplus.com/articles/the-beginners-guide-to-unit-testing-what-is-unit-testing--wp-25728

Three reasons for unit testing (3)

- Since you're testing your code as you introduce your functionality, you can prevent changes and additions from breaking functionality

http://code.tutsplus.com/articles/the-beginners-guide-to-unit-testing-what-is-unit-testing--wp-25728

Test terminology

- *Unit test (test case)* – tests a single method (unit of functionality)
- *Integration test* - tests the behavior of a component or the integration between a set of components
- *Performance tests* - used to benchmark software components repeatedly. This is to ensure that the code runs fast enough even if it’s under high load
- *Mock(ing)* - a real object is exchanged by a replacement which has a predefined behavior for the test (DAO)

Part two

Creating tests with JUnit

JUnit testing

- JUnit is the Java framework for creating unit tests and test suites
- You find it at www.junit.org
- We will be working with JUnit5, the most recent version of this framework

To get started with JUnit5 (and Gradle>4.5): an example Gradle build script

```
group 'nl.bioinf.junit5tests'
version '1.0-SNAPSHOT'

apply plugin: 'java'
apply plugin: 'idea'

sourceCompatibility = JavaVersion.VERSION_1_10

repositories {
    mavenCentral()
}

dependencies {
    testImplementation 'org.junit.jupiter:junit-jupiter-api:5.1.0'
    testRuntimeOnly 'org.junit.jupiter:junit-jupiter-engine:5.1.0'
    //parameterized tests
    testImplementation 'org.junit.jupiter:junit-jupiter-params:5.1.0'
}
```

The essence of testing

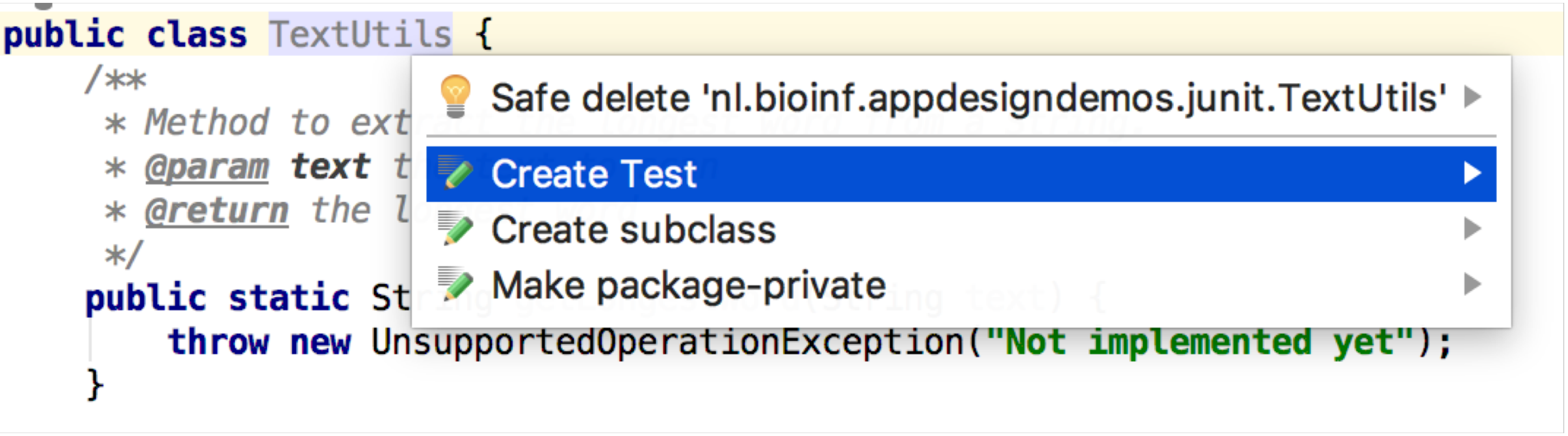
- Suppose you are creating a class, **`TextUtils`** with method **`getLongestWord(String text)`**
- This is how you it looks like without any functionality implemented:

My TextUtils class stub

```java
package nl.bioinf.junit5tests;

public class TextUtils {
    /**
     * Finds the longest word in a given string.
     * If there is a tie, the last found longest
     * is returned.
     *
     * @param text the text to scan
     * @return longestWord
     */
    public static String getLongestWord(String text) {
        throw new UnsupportedOperationException(
                                      "Not implemented yet");
    }
}
```

Create the test class

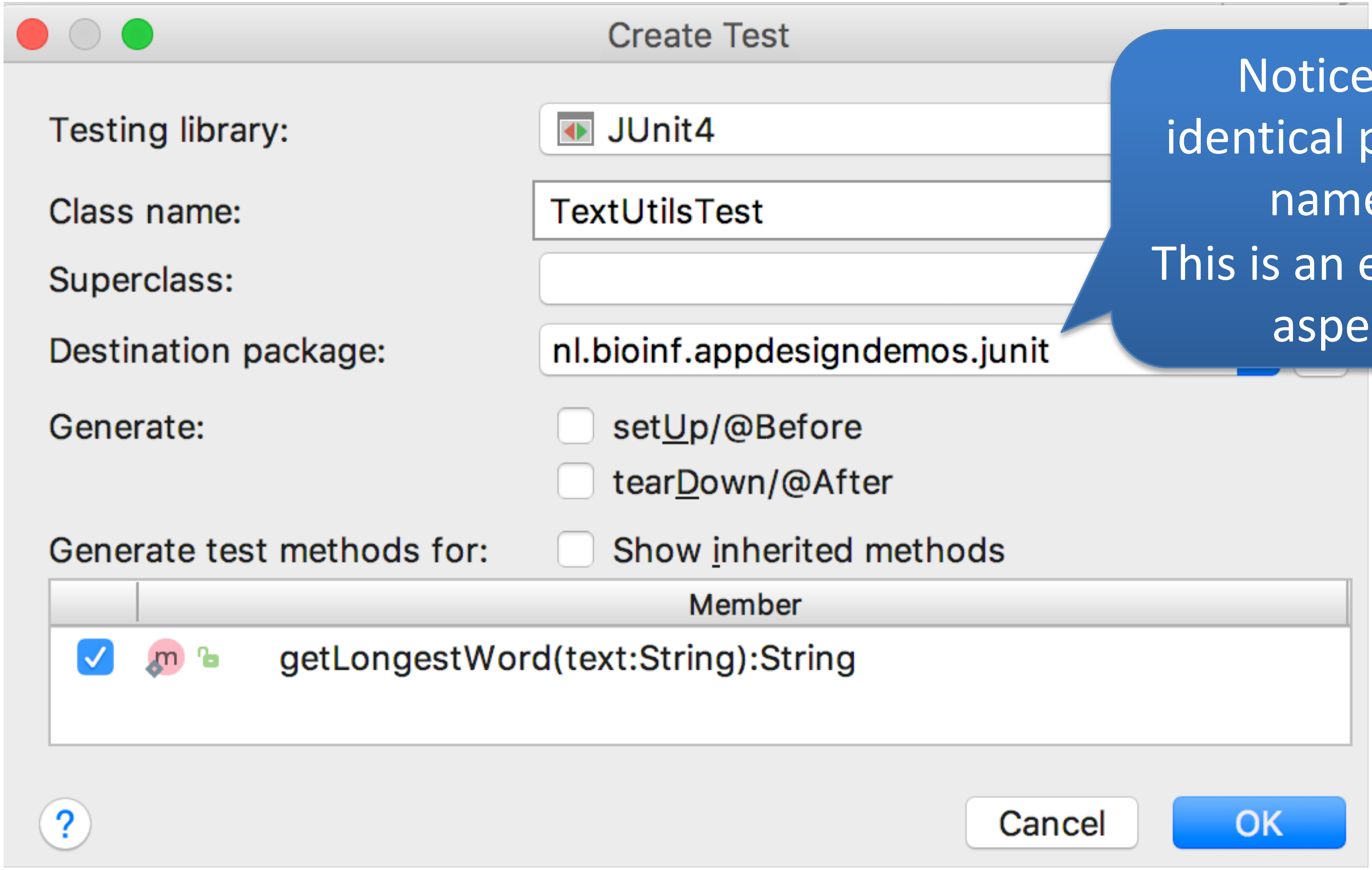
- In IntelliJ, you can select the class name, press (Alt) + enter in the editor and select "Create Test”



```
public class TextUtils {
    /**
     * Method to ext
     * @param text t
     * @return the l
     */
    public static St
        throw new UnsupportedOperationException("Not implemented yet");
    }
```

- This will give you the following dialog

class TextUtilsTest



Create Test

Testing library: JUnit4

Class name: TextUtilsTest

Superclass:

Destination package: nl.bioinf.appdesigndemos.junit

Generate: □ setUp/@Before

□ tearDown/@After

Generate test methods for: □ Show inherited methods

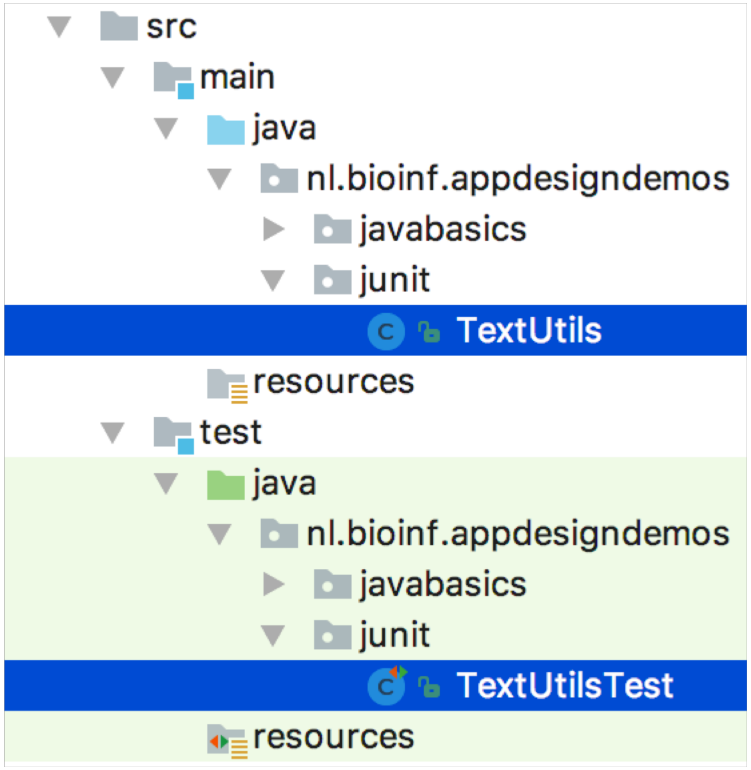
	Member
☑	getLongestWord(text:String):String

Notice the identical package names!

This is an essential aspect.

class TextUtilsTest

- In you project explorer, you'll see the newly created test



```
▼ src
    ▼ main
        ▼ java
            ▼ nl.bioinf.appdesigndemos
                ▶ javabasics
                ▼ junit
                        TextUtils
            resources
    ▼ test
        ▼ java
            ▼ nl.bioinf.appdesigndemos
                ▶ javabasics
                ▼ junit
                        TextUtilsTest
            resources
```

Why the identical package names?

- Since implementation and test code are in the same package scope, all non-private methods from the class to test are visible:
  - **public**
  - **protected**
  - **package-public (default)**

class TextUtilsTest – first version

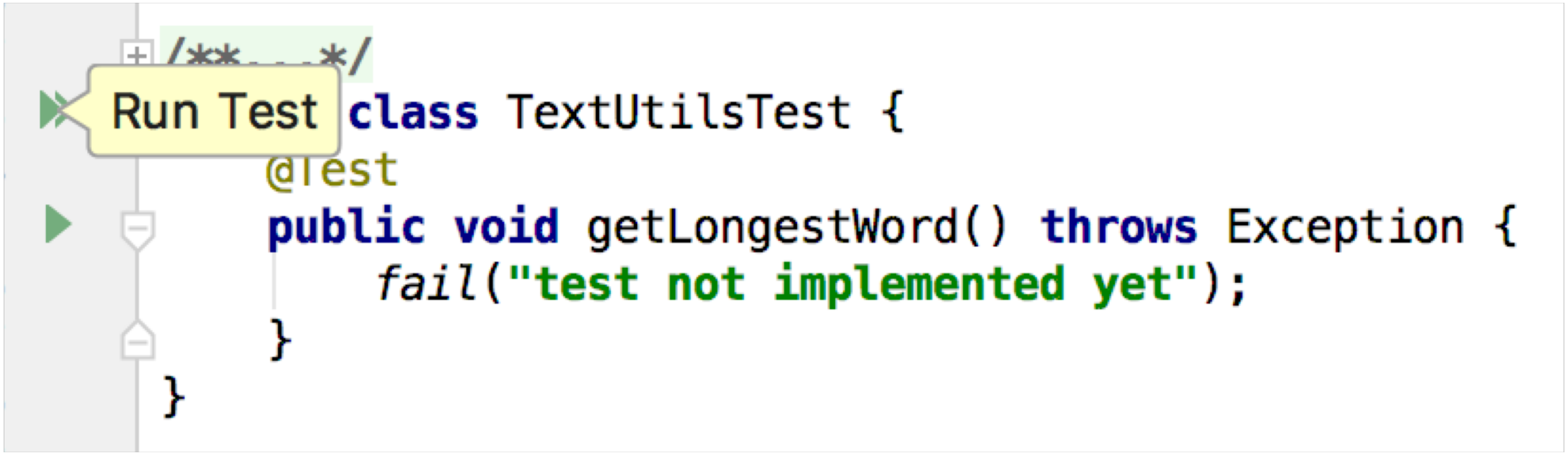
```java
package nl.bioinf.appdesigndemos.junit;

import org.junit.jupiter.api.*;

public class TextUtilsTest {
    @Test
    public void getLongestWord() throws Exception {
        fail("test not implemented yet");
    }
}
```

Except for the underlined statement, this is all auto-generated code!

Of course it should fail by default!

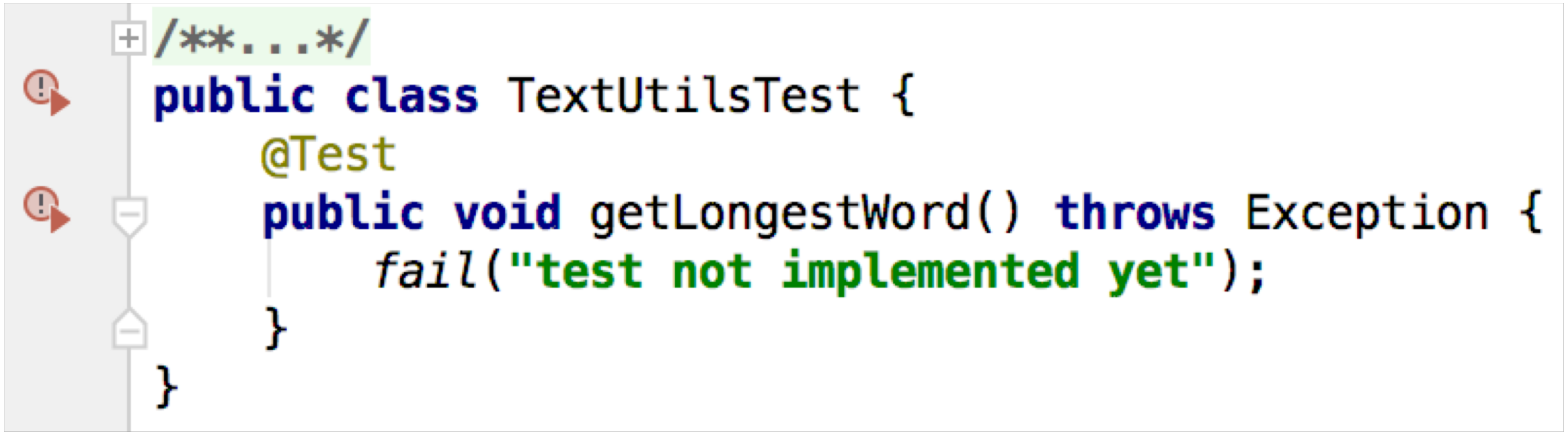
Run the test

- In the editor margin, you see arrows
- These are the simplest way to run one or more tests (the entire test class or a single case)

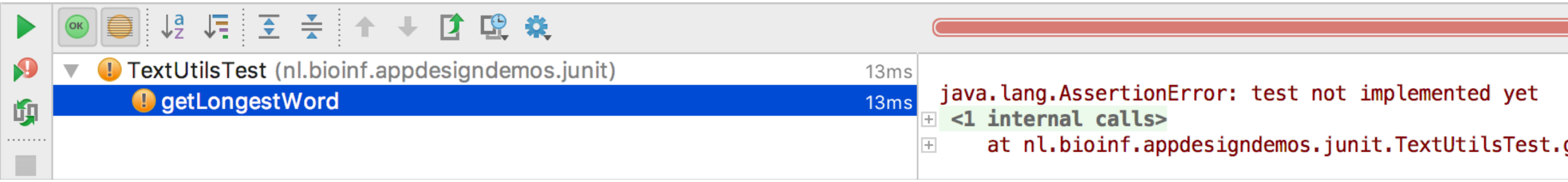


```
/**...*/
class TextUtilsTest {
    @Test
    public void getLongestWord() throws Exception {
        fail("test not implemented yet");
    }
}
```

Run the test...and fail



```
/**...*/
public class TextUtilsTest {
    @Test
    public void getLongestWord() throws Exception {
        fail("test not implemented yet");
    }
}
```



```
▼ TextUtilsTest (nl.bioinf.appdesigndemos.junit)   13ms
    getLongestWord                                 13ms
```

```
java.lang.AssertionError: test not implemented yet
<1 internal calls>
    at nl.bioinf.appdesigndemos.junit.TextUtilsTest.g
```

- Of course it fails – you'll need to create some test logic!

class TextUtilsTest – second version

- This is a good place to start – a so-called “sunny day” scenario

```
public class TextUtilsTest {
    @Test
    public void getLongestWord() throws Exception {
        String inputText = "Hello JUnit testing world";
        String expResult = "testing";
        String result = TextUtils.getLongestWord(inputText);
        assertEquals(expResult, result);
    }
}
```

class TextUtils – second version

- Now implement the logic to pass the first test case

```
public static String getLongestWord(String text) {
    String[] words = text.split(" ");
    String longest = "";
    for (String word : words) {
        if (word.length() >= longest.length()) {
            longest = word;
        }
    }
    return longest;
}
```

class TextUtils – second Java8 version

- I read about lambdas so have a go at this too

```java
public static String getLongestWord(String text) {
    String[] words = text.split(" ");
    Optional<String> findFirst =
            Stream.of(words)
                    .sorted((one, two)->
        Integer.compare(two.length(), one.length()))
                    .findFirst();
    return findFirst.get();
```

It passes! I’m done…am I?

- Of course not!
- Writing test cases for *sunny day* scenarios only is not enough

Expand the test case repertoire

- Take a minute to think about some useful tests that will demonstrate the reliability of the code
- Take extra care of *boundary cases!*
- Should you maybe even change the signature of the method `getLongestWord()`?

Extended conditions

- But what about these inputs?
  - `"The quick brown fox jumps over the lazy dog"`
  - `""`
  - `" "`
  - `"foo bar 1234"`
  - *`null`*
- What should your method do, and the test case expect?
- This is one of the main aspect of unit testing – it makes you think about special and boundary cases

Assert methods

- These are the Assert methods you can use in your tests. Their names are pretty much self explanatory:

```
assertArrayEquals()
assertEquals()
assertTrue() & assertFalse()
assertNull() & assertNotNull()
assertSame() & assertNotSame()
assertThat()
```

There are overloaded variants for each of them

Extended conditions

  - `"The quick brown fox jumps over the lazy dog"`
  - `""`
  - `"  "`
  - `"foo bar 1234"`
  - *`null`*
- Now take some time to implement tests and functionality covering these method use cases

Document!

- Very important - don’t forget to describe decisions in the Javadoc section:

```
/**
 * This method searches for the longest word in
 * a given string.
 * It will split the String on all spaces and
 * removes all non-word characters (matching
 * the pattern "[^A-Za-z]". If multiple words
 * have the same length, it will return the
 * first of these
 * @param text the text to analyze
 * @return longest the longest word
 * @throws IllegalArgumentException ex when
 *    a null value or empty string is passed
 */
```

JUnit5 annotations

- You should have seen ***annotations*** in your Java code by now:
  - `@Override`
  - `@SuppressWarnings("unused")`
- Annotations are form of syntactic metadata that can be added to Java source code (classes, methods, variables)
- This is the main technique for JUnit testing

JUnit annotations: @Test

- @Test is the most important annotation: it indicates the method is a JUnit test and should be run as such

```
@Test
public void testImportant(){
    String first = "Michiel";
    String second = "Michiel";
    assertSame("these should be " +
     " the same object", first, second);
}
```

You notice something funny in this test? Will it pass or fail?

JUnit annotations: @Disabled

- @Disabled (JUnit4: @Ignore) is used to (temporarily) disable a test

```
@Test
@Disabled
public void testImportant(){
    String first = "Michiel";
    String second = "Michiel";
    assertSame("these should be "
        + " the same object", first, second);
}
```

JUnit annotations: @BeforeEach and @AfterEach

- @BeforeEach (@Before in <JUnit5) and @AfterEach (@After) run respectively before and after ***each test case***.

```
private String defaultSentence;

@BeforeEach
public void setUp() {
    defaultSentence = "My favorite
      programming language is Java";
}
```

defaultSentence will get its value back after each test

JUnit annotations: @BeforeAll and @AfterAll

- @BeforeAll (JUnit 4: @BeforeClass) and @AfterAll (JUnit4: @AfterClass) annotations are similar to @AfterEach and @BeforeEach with the difference that they are called once per TestClass and not on per test basis and can be used to initialize class level resources.
- These methods should be *static*.

Testing for Exceptions

-
  ```
  assertThrows(
          Class<? extends Throwable>
  expectedType,
          Executable executable)
  ```
  in JUnit5 is used to assert that the supplied executable will throw an exception of the expectedType.
- It relies on lambdas

```
@Test
void shouldThrowException() {
    Throwable exception = assertThrows(
                IllegalArgumentException.class,
                () -> TextUtils.getLongestWord(null));
    assertEquals(exception.getMessage(), "text cannot be null");
}
```

Testing for Exceptions

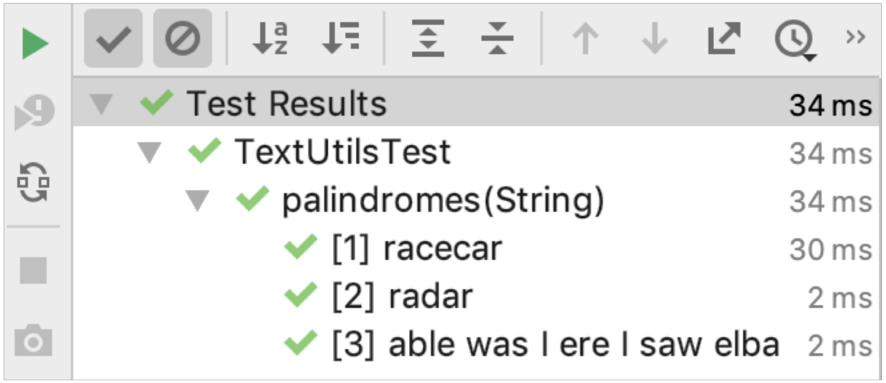
- Or use plain old Java (JUnit 4 strategy)

```
@Test
public void shouldThrowExceptionOldSchool() {
    try {
        TextUtils.getLongestWord(null);
        fail("Expected an IllegalArgumentException");
    } catch (IllegalArgumentException e) {
        assertEquals("text cannot be null", e.getMessage());
    }
}
```

Parameterized tests

- To repeat a test with differing inputs

```
@ParameterizedTest
@ValueSource(strings =
    {"racecar",
     "radar",
     "able was I ere I saw elba"})
void palindromes(String candidate) {
    assertTrue(isPalindrome(candidate));
}
```



```
▼ ✓ Test Results                          34 ms
   ▼ ✓ TextUtilsTest                      34 ms
      ▼ ✓ palindromes(String)             34 ms
           ✓ [1] racecar                  30 ms
           ✓ [2] radar                     2 ms
           ✓ [3] able was I ere I saw elba 2 ms
```

There’s more

- You should really have a look at http://junit.org/junit5/docs/current/user-guide/
  to get an idea of some really cool tricks you can do, such as
  - Extensions (formerly Rules)
  - Test Suites
  - And much more

Part two

What and how to test

What to test

- …
- Uhhh sorry, we couldn't agree

- **You should write software tests for the critical and complex parts of your application**

Test strategy

- Create tests for
  - common usage
  - empty sets
  - null arguments
  - conflicting arguments
  - illegal arguments (e.g. create list of $1*10^{21}$ elements)

Test strategy example

- For a function which is supposed to take two parameters and should return a value after doing some processing, then different use cases might be:
  - First parameter can be null. It should throw an IllegalArgumentException.
  - Second parameter can be null. It should throw an IllegalArgumentException
  - Both can be null. It should throw an IllegalArgumentException
  - Finally, test the valid output of function. It should return valid pre-determined output.

Test rules

- Test only one code unit at a time
- Make each test independent of all others
- Mock out all external services and state and don’t test configuration settings
- Write tests for methods that have the fewest dependencies first, and work your way up
- Use the most appropriate assertion methods
- Ensure that test code is separated from production code
- Do not print anything out in unit tests

http://howtodoinjava.com/2012/11/05/unit-testing-best-practices-junit-reference-guide/

Testing private methods (1)

- Testing private methods is something that has traditionally drawn heated debates in the Java world.
- The solutions usually fall into 4 categories:
  - Don't test private methods
  - Use reflection
  - Use a nested class
  - Change the visibility
- Let's look at them in turn

these slides from http://axelfontaine.com/blog/private-methods.html

Testing private methods (2)

Don't test private methods

- This really leaves us with three choices:
  - refactor to make the method public in some helper class
  - test through a calling method with a higher visibility
  - give up
- A delightful choice between increased bloat, higher test complexity and resignation!
- No, thank you!

Testing private methods (3)

Use reflection

- Why make things simple when you can also make them hard and long-winded?

- No, thank you!

- We won't even go into what reflection is

Testing private methods (4)

Use a nested class

- No too bad, but with 3 significant drawbacks:
  - no separate sources / test sources folders
  - larger classes
  - unit test code in production binaries
- We can do better.

- No thank you!

Testing private methods (5)

Change the visibility

- The last option. Not perfect either, but by far the most pragmatic!
- It trades a slight increase in visibility (to package-protected) for greatly simplified calling (a regular method call) while still preserving the sources / test sources separation.
- And with a simple documentation habit, it becomes clear to everyone why this design trade-off was made:

```
/* private -> testing */ void
myMethodUnderTest() {   ...   }
```