# Practice 3: JUnit

## Software Design (614G01015)

**Eduardo Mosqueira Rey** (Coordinador)

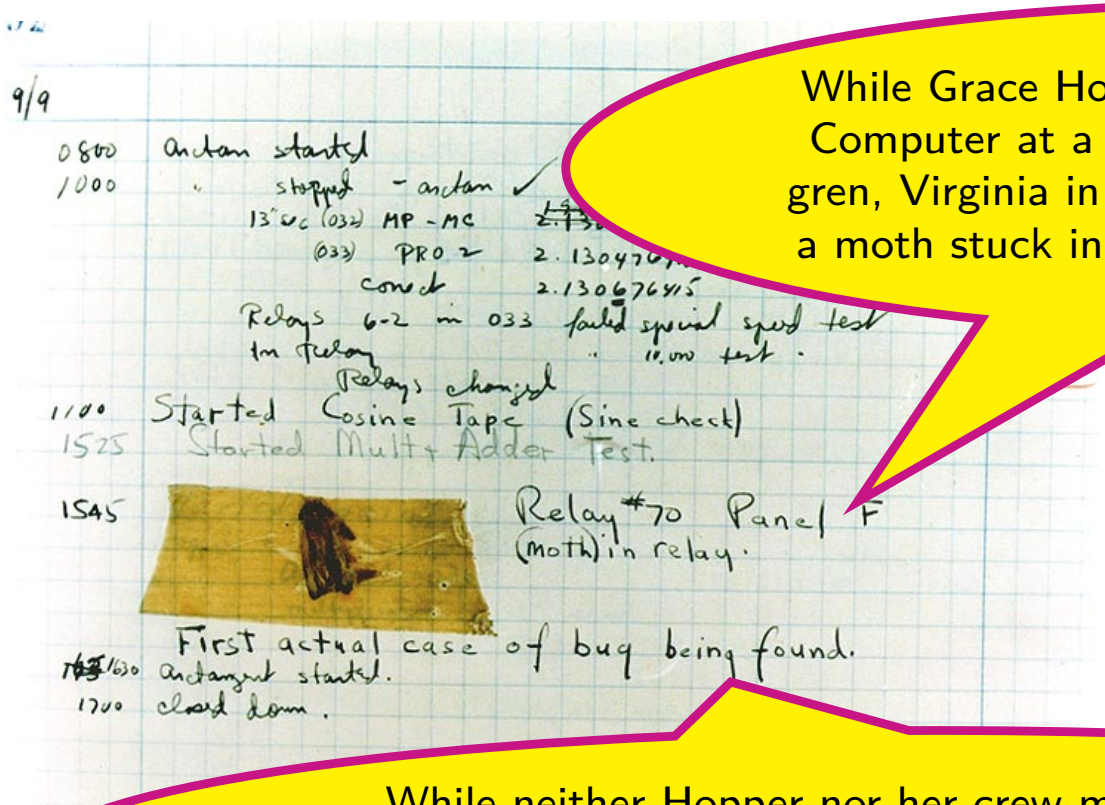
Department of Computer Science and Information Technology
Faculty of Computer Science

UNIVERSIDADE DA CORUÑA

# Table of Contents

# Table of Contents

# Introduction

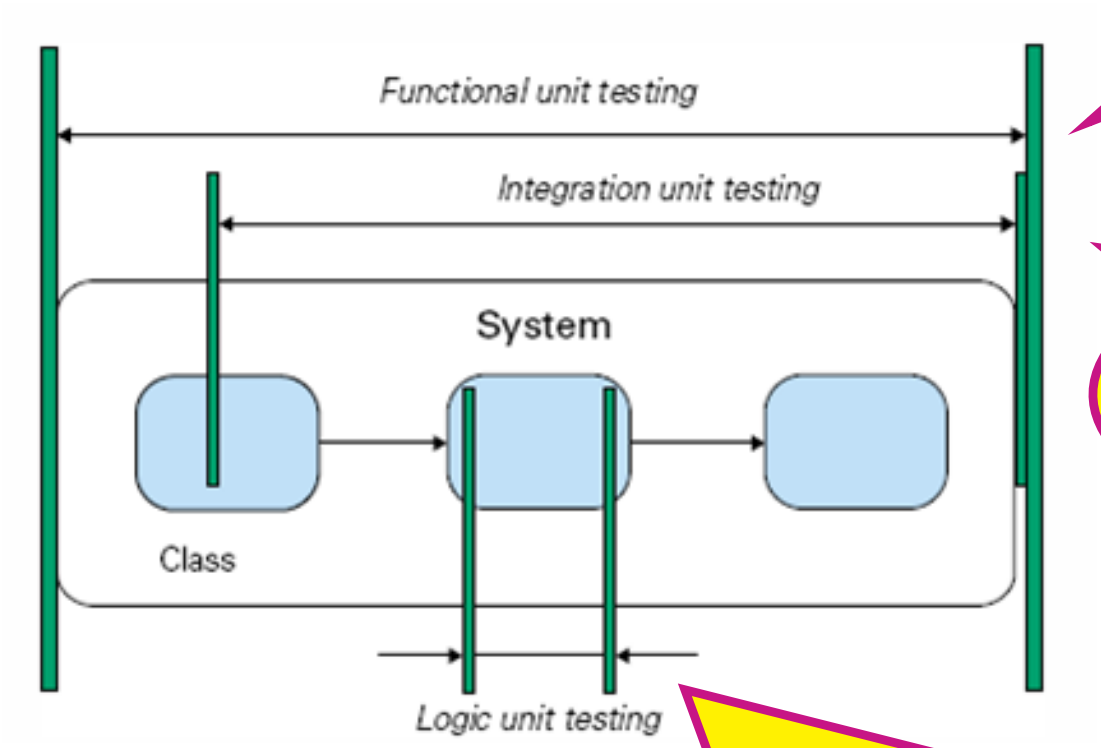



While Grace Hopper was working on a Mark II Computer at a US Navy research lab in Dahlgren, Virginia in 1947, her associates discovered a moth stuck in a relay impeding its operation.

While neither Hopper nor her crew mentioned the phrase "debugging" in their logs, the case was held as an instance of literal "debugging", perhaps the first in history (Wikipedia).

# Introduction

The scope of functional tests is the whole system. They test functional requirements from the specification.




The scope of integration tests is the interaction between components (especially external, e.g., a DB).

The scope of unit tests is a class.

# Table of Contents

# Test-Driven Development

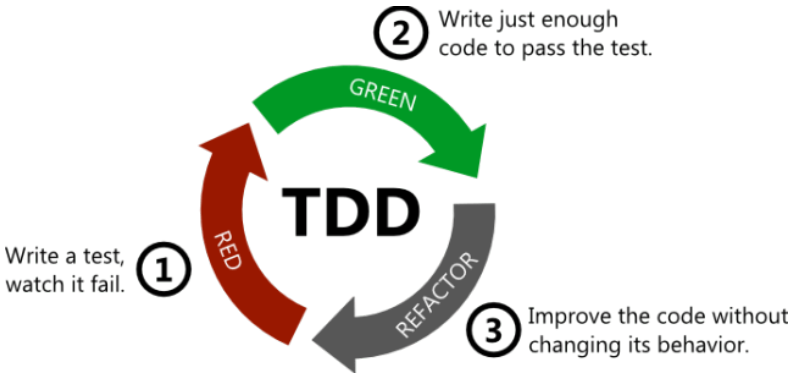
**Test-Driven Development, TDD**

A programming methodology where you write the tests BEFORE you write the actual code.

- It belongs to agile methodologies.
- The rationale is that writing the tests at the beginning facilitates writing and debugging the subsequent code.
- It also facilitates its *refactoring* (i.e., rewriting its internal implementation).

# Typical outline in TDD

1. Write a stub for the class you want to create. That is, define the methods but do not write the actual implementation yet.
2. Design and write the tests for those methods and make sure that the tests fail (because the original methods have not been implemented yet).
3. Now write the actual implementation for the original methods.
4. Run the tests again. Your methods should now pass the tests (assuming the tests were well designed). Otherwise, check what has failed.
5. Good code can always be refactored in order to optimize it. Your new methods should still pass your old tests.



# Benefits of TDD

- **Documenting code**
  - The tests represent the requirements that our code must meet.
- **Focusing on observable behavior, rather than implementation**
  - Writing the tests at the beginning means adopting the point of view of an external client. This prioritizes interface aspects over implementation aspects.
- **Facilitating the detection of errors**
  - If the tests are well desinged, checking the validity of the code is as easy as running the tests. This simplifies the task of debugging.
- **Automating the detection of errors**
  - If we automate testing and then refactor our application, it's easy to check that everything still works correctly. This makes us feel more comfortable with refactoring.

# Table of Contents

# JUnit

JUnit 5

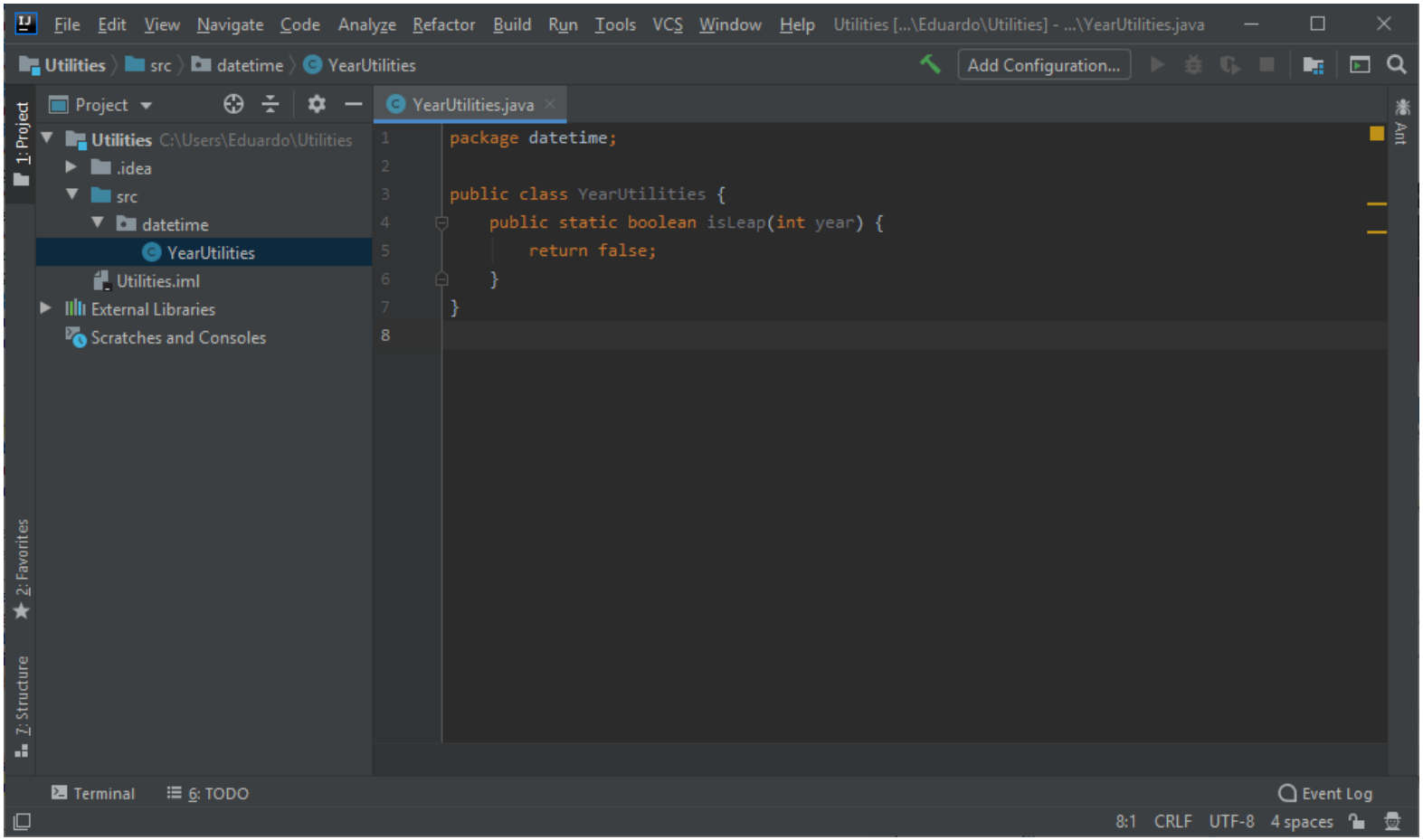
- JUnit is an open-source Java library that helps us develop unit tests (moreover, it can also be used for integration tests or functional tests).
- Before these kind of tools, unit tests were written directly in main methods (awkward and rigid).
- IntelliJ IDEA integrates the JUnit library, so creating a test for a given class is easy.
- More information at: `https://junit.org/junit5/` and `https://www.jetbrains.com/help/idea/configuring-testing-libraries.html`

# Example

- **Goal**
  - Writing a function that determines if a given year is a leap year.
- **Specifications**
  - A leap year is divisible by four (i.e., every four years we get a leap year).
  - Exception 1: The years that are divisible by 100 are not considered leap years (every 100 years, there is a year that should be a leap year but isn't, e.g., the year 1900).
  - Exception 2: But if the year is divisible by 400, then it *is* a leap year (e.g., the year 2000).

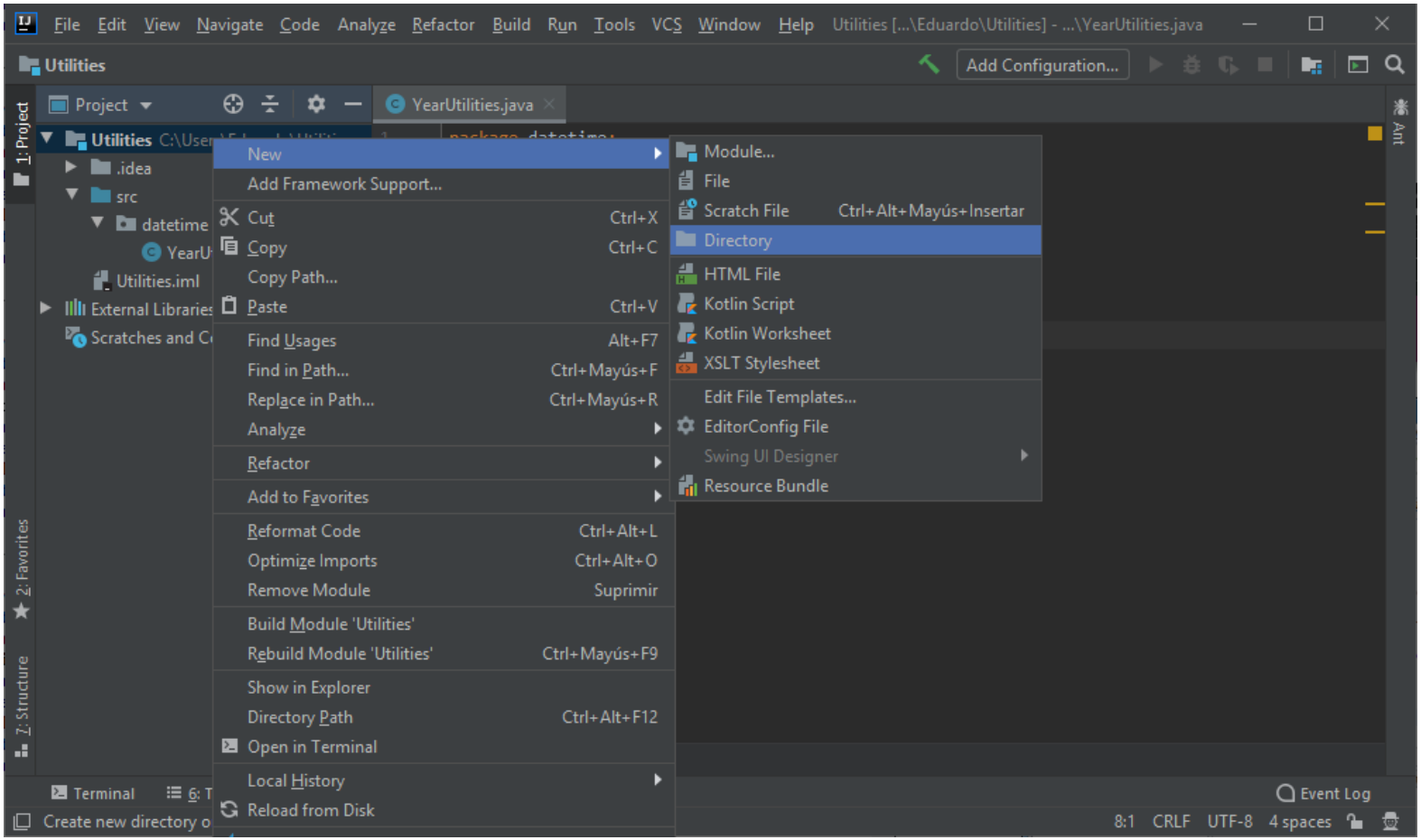
# Example

- The first step is creating a trivial version of the code to be developed.



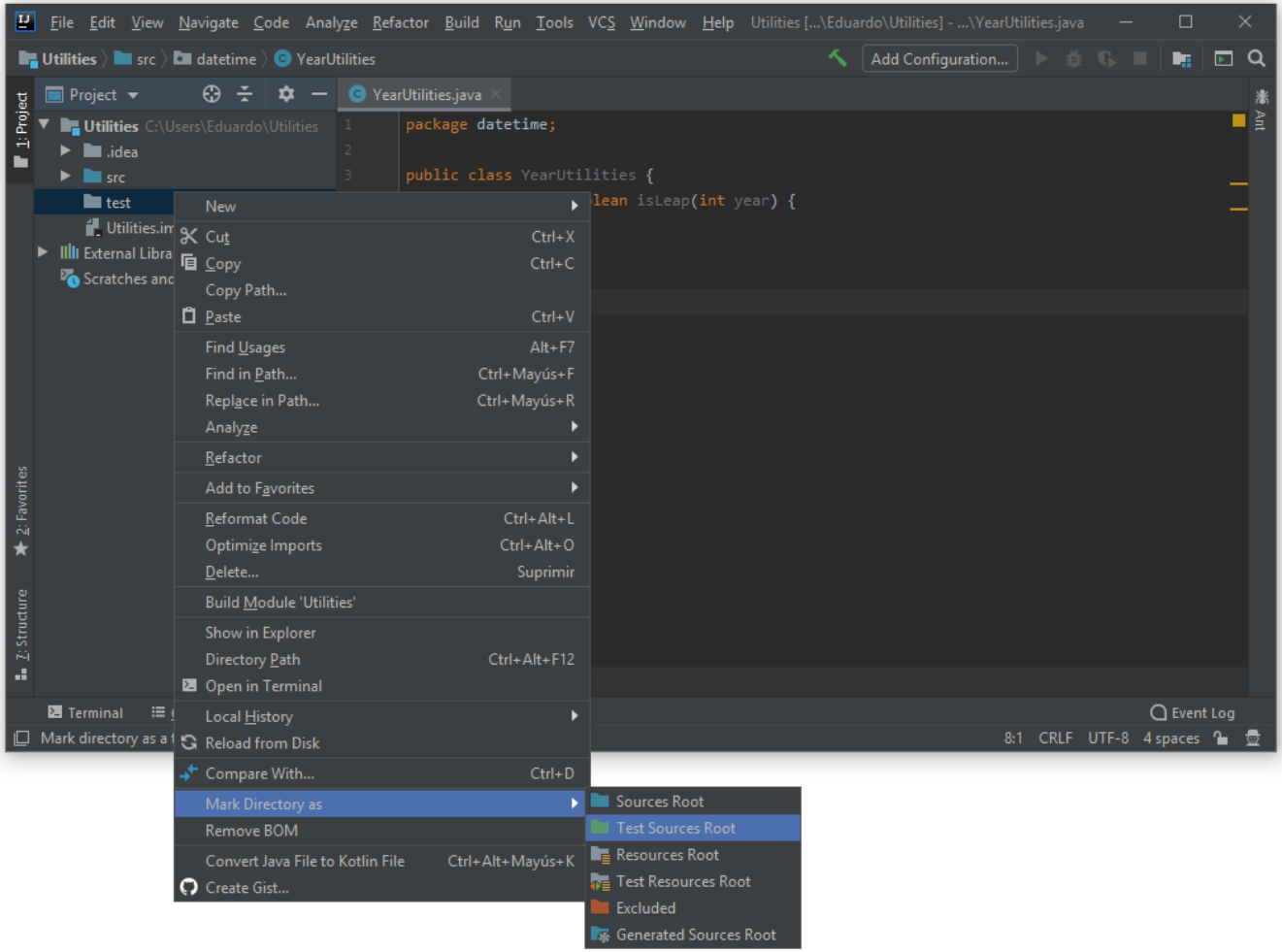
# Example

- Before we create the tests, we must create a `test` folder to save them (we recommend keeping them separate from the source code).



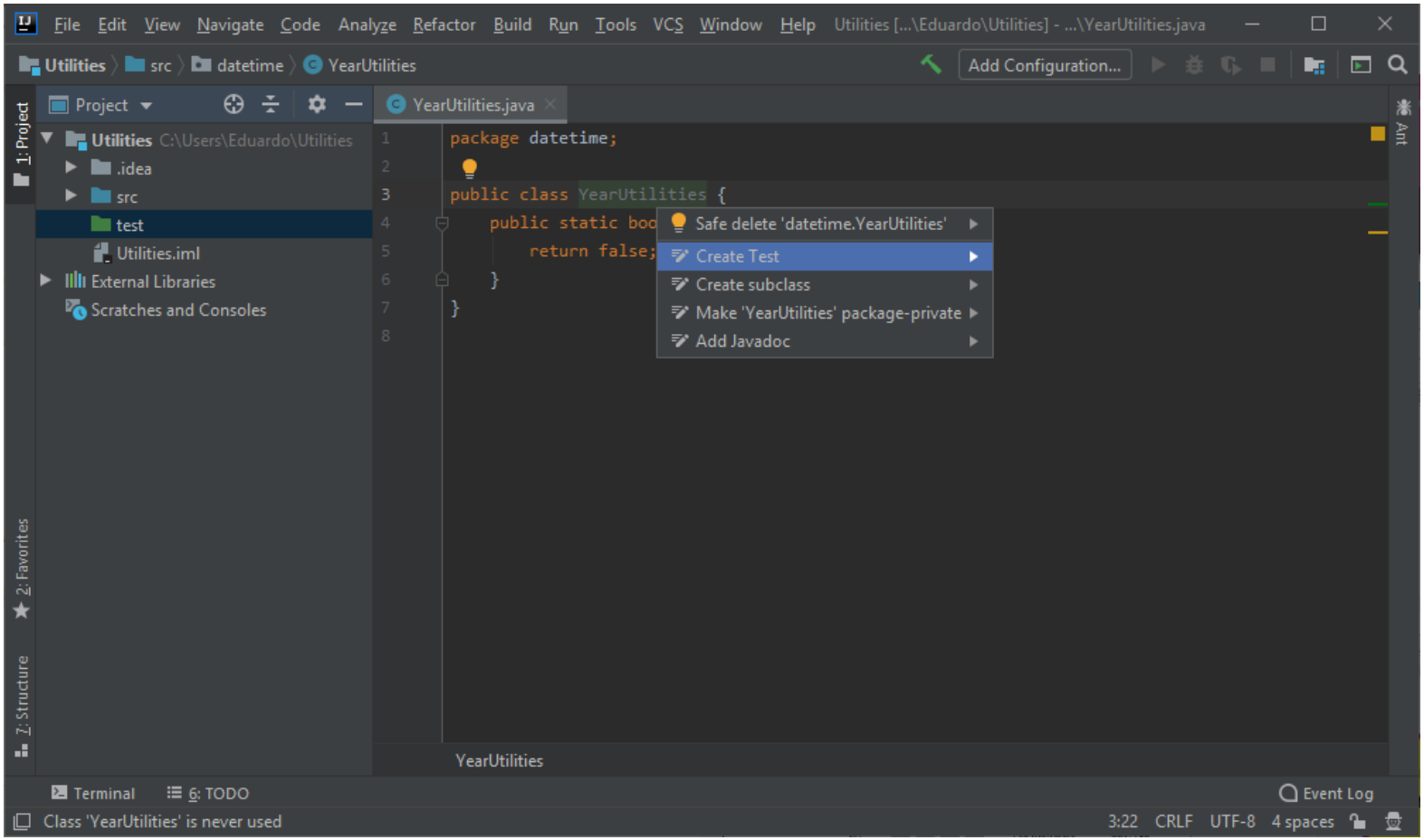
# Example

- It is necessary to tell IntelliJ IDEA that the `test` folder contains the tests.



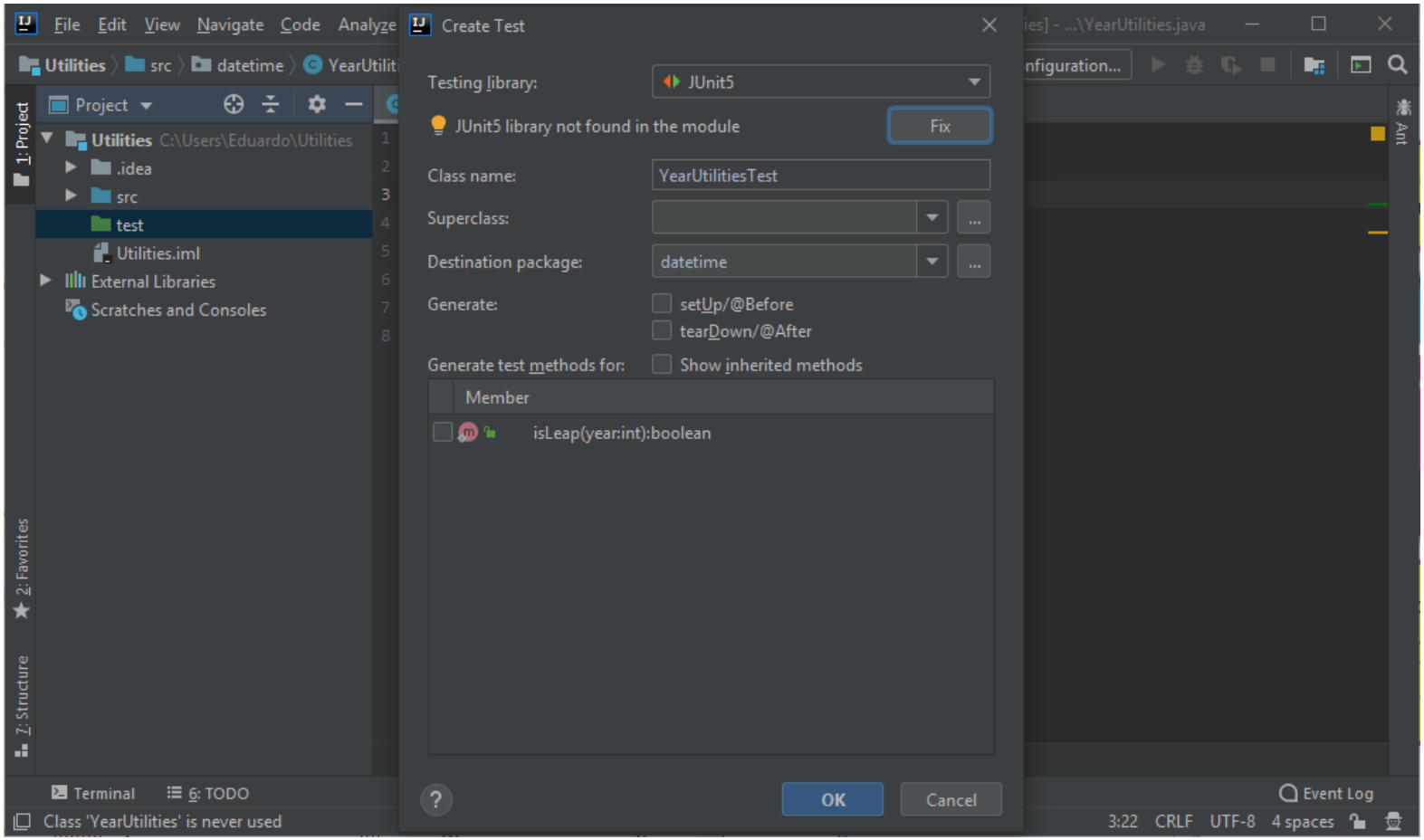
# Example

- To create a test for a specific class, hover over the class name, hit `Alt+Tab` and select *Create Test*.



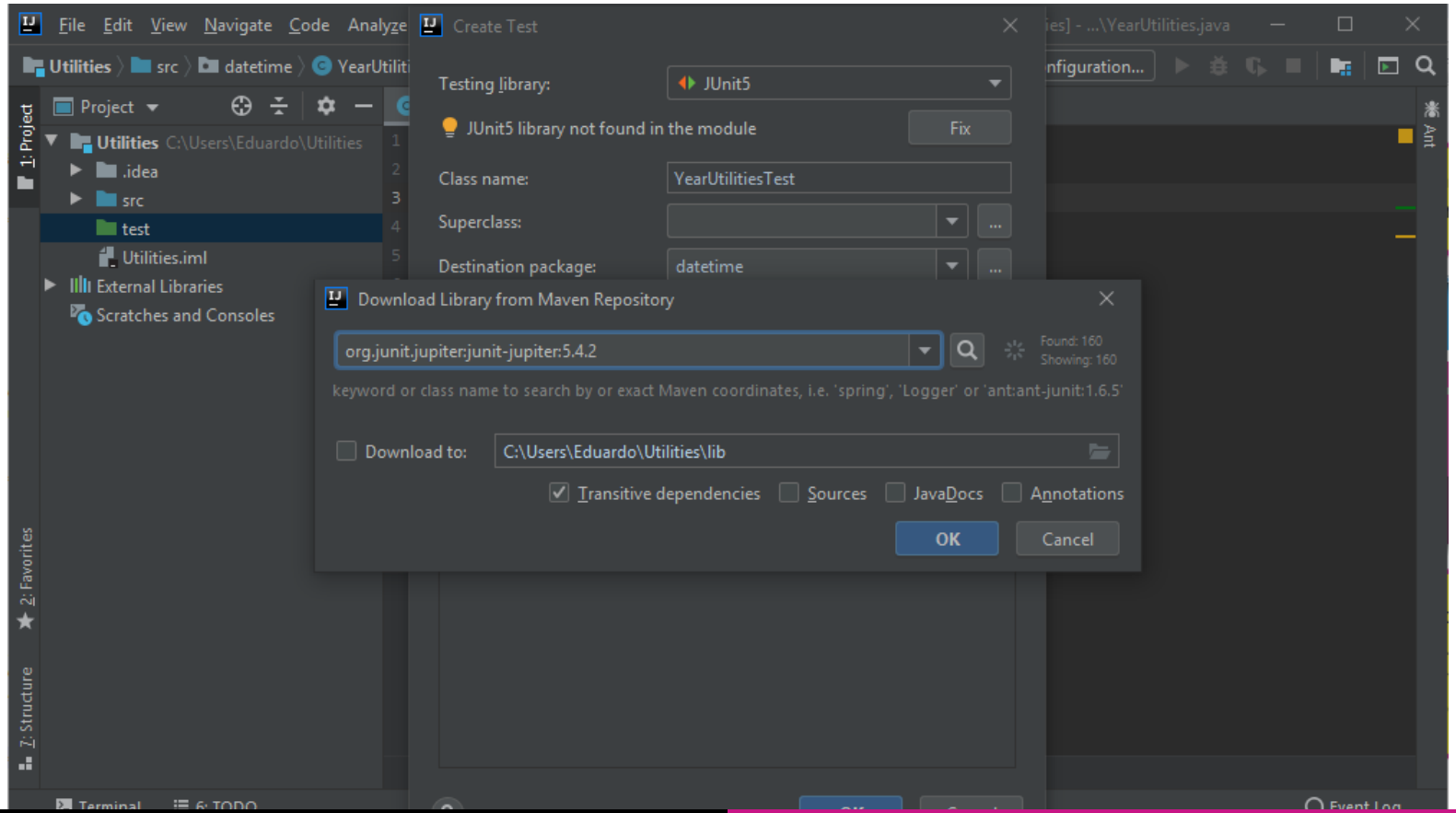
# Example

- Choose the JUnit 5 library. Since it is not included in the current module, you need to click on the *Fix* button.



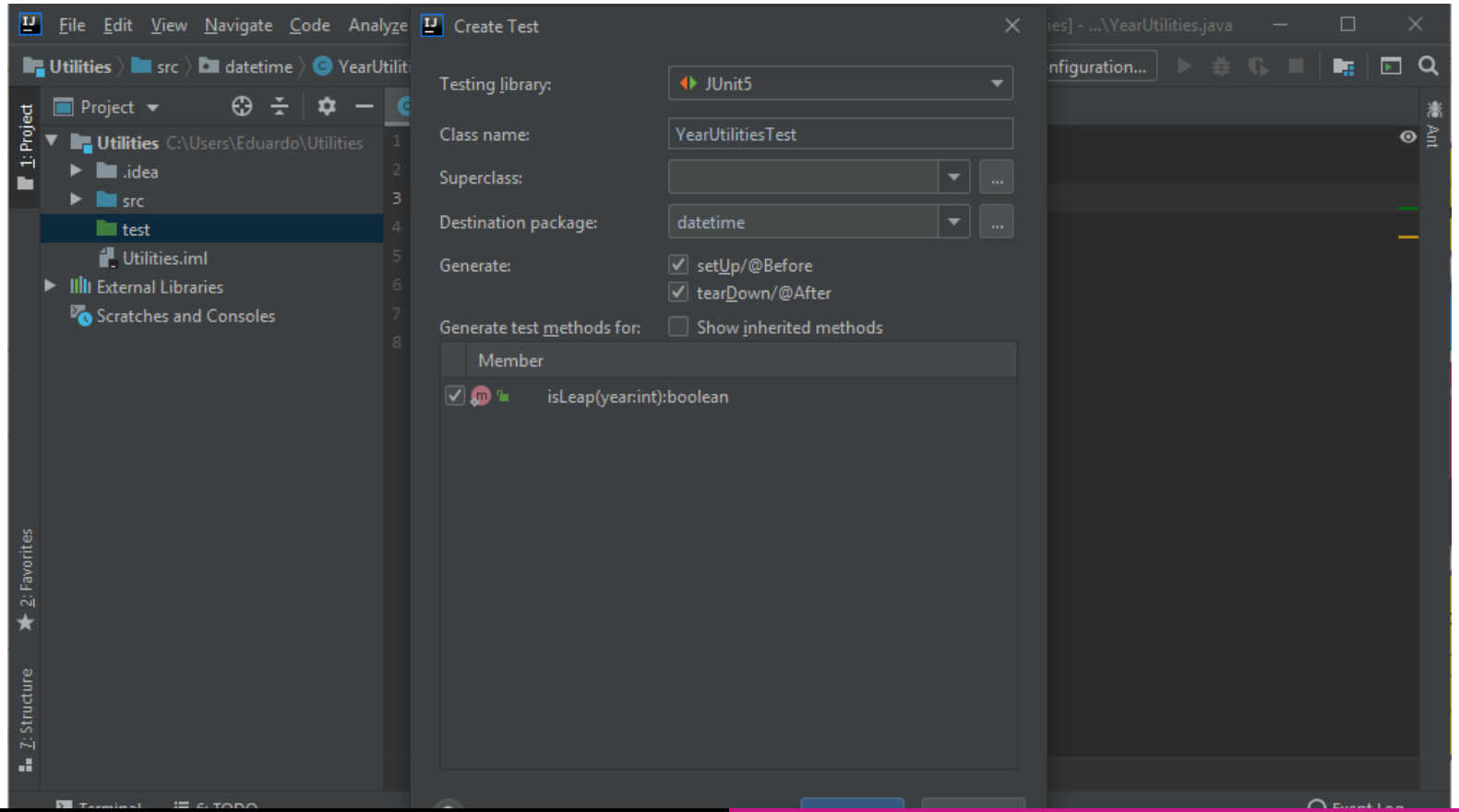
# Example

- IDEA will download the library from a Maven repository. By default, it will be included in the IntelliJ IDEA distribution (recommended), although it is possible to add it only to the `lib` folder of the current project.



# Example

- We choose the name for the test class (the original name with the `test` suffix), the package that contains it (the original one), whether we include `@Before` and `@After` methods, and the method being tested (i.e., `isLeap`).



# Example

- The external JUnit library is created, the `YearUtilitiesTest` class is created with methods tagged with "`@`" annotations, and `junit` packages are imported.

```
import org.junit.jupiter.api.AfterEach;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;

import static org.junit.jupiter.api.Assertions.*;

class YearUtilitiesTest {

    @BeforeEach
    void setUp() {
    }

    @AfterEach
    void tearDown() {
    }

    @Test
    void isLeap() {
    }
}
```

# Summary of the main JUnit 5 annotations

| Annotation[1] | Meaning |
|---|---|
| `@Test` | Marks the method as a test. The test succeeds if it finishes and no exceptions were thrown. |
| `@BeforeEach` | Indicates that the method must be run **before each** test method from the class. It is generally used to create instances that will be later shared by all the tests. |
| `@AfterEach` | Indicates that the method must be run **after each** test method from the class. Its execution is guaranteed even if the `Before` or `Test` methods are interrupted by an exception. |
| `@BeforeAll` | Indicates that the method must be run **before all** the test methods from the class. |
| `@AfterAll` | Indicates that the method must be run **after all** the test methods from the class. Its execution is guaranteed even if the `BeforeAll` or `Test` methods are interrupted by an exception. |
| `@Disabled` | Marks a test as temporarily disabled. |
| `@Timeout(1)` | Makes a test fail if its execution takes longer than a specified duration. The duration is expressed in seconds by default, but that is configurable. |

[1] The full list can be consulted at `https://junit.org/junit5/docs/current/user-guide/#writing-tests-annotations`

# Example

- We use JUnit `assert` methods to check that our code works as expected.
- We must aim to cover all possible situations.

```
package datetime;

import org.junit.jupiter.api.AfterEach;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;

import static org.junit.jupiter.api.Assertions.*;

class YearUtilitiesTest {

    @Test
    void isLeap() {
        assertFalse(YearUtilities.isLeap( year: 2019)); // not leap, not divisible by 4
        assertTrue(YearUtilities.isLeap( year: 2020)); // leap, divisible by 4
        assertFalse(YearUtilities.isLeap( year: 1900)); // not leap, divisible by 4 and 100
        assertTrue(YearUtilities.isLeap( year: 2000)); // leap, divisible by 4, 100 and 400
    }
}
```

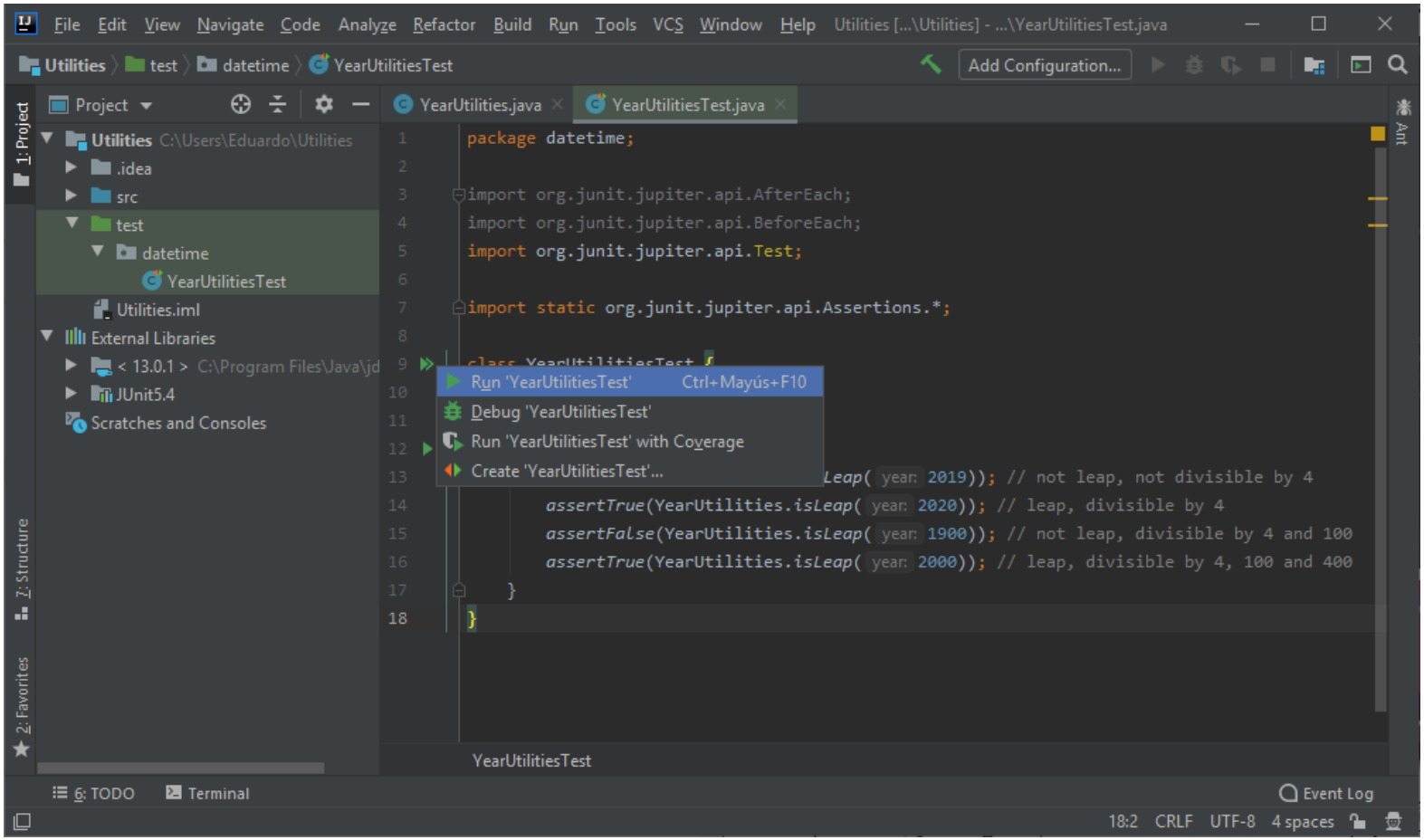
# Summary of the main JUnit 5 assertions

| Method[2] | Meaning |
|---|---|
| `assertArrayEquals (type[] expected, type[] actual)` | Compares two arrays with `type` = `boolean`, `char`, `byte`, `short`, `int`, `long` or `Object`. |
| `assertArrayEquals (type[] expected, type[] actual, double delta)` | Compares two arrays of decimal numbers, with a margin of error of `delta`. `type` = `double`, `float`. |
| `assertEquals (type expected, type actual)` | Checks if two elements are equal. `type` = `char`, `byte`, `short`, `int`, `long`, `Object`. |
| `assertEquals (type expected, type actual, double delta)` | Checks if two decimal numbers (`type` = `float`, `double`) are equal, with a margin of error of `delta`. |
| `assertFalse(boolean condition)` | Checks if the specified condition is false. |
| `assertTrue(boolean condition)` | Checks if the specified condition is true. |
| `assertNotNull(Object object)` | Checks that an object is NOT null. |
| `assertNull(Object object)` | Checks that an object is null. |
| `assertNotSame (Object unexpected, Object actual)` | Checks that two objects do NOT refer to same object (identity). |
| `assertSame (Object expected, Object actual)` | Checks that two objects do refer to same object (identity). |
| `assertThrows(Class<T> expectedType, Executable executable)` | Checks that the given code throws the specified exception. |
| `fail()` | Forces a test to fail. |

[2] The full list of methods can be consulted at `https://junit.org/junit5/docs/current/api/org/junit/jupiter/api/Assertions.html`

# Example

- We run all the tests in a class by clicking on the double green triangle, or hitting `Ctrl+Shift+F10`.
- Or you can run just a particular test by clicking on its green triangle.



# Example

- When you run the tests, you can see how some fail.
- Those that fail are highlighted in red and the corresponding line of code is underlined.

```
@Test
void isLeap() {
    assertFalse(YearUtilities.isLeap( year: 2019)); // not leap, not divisible by 4
    assertTrue(YearUtilities.isLeap( year: 2020)); // leap, divisible by 4
    assertFalse(YearUtilities.isLeap( year: 1900)); // not leap, divisible by 4 and 100
    assertTrue(YearUtilities.isLeap( year: 2000)); // leap, divisible by 4, 100 and 400
}
}
```

```
org.opentest4j.AssertionFailedError: expected: <true> but was: <false>
<4 internal calls>
    at datetime.YearUtilitiesTest.isLeap(YearUtilitiesTest.java:14) <19 internal calls>
    at java.base/java.util.ArrayList.forEach(ArrayList.java:1507) <9 internal calls>
```

# Example

- You can now write the full code for the class to check if it passes the tests.

```
package datetime;

public class YearUtilities {
    public static boolean isLeap(int year) {
        if (year % 4 == 0)
            if (year % 100 == 0)
                if (year % 400 == 0)
                    return true;
                else return false;
            else return true;
        else return false;
    }
}
```

# Example

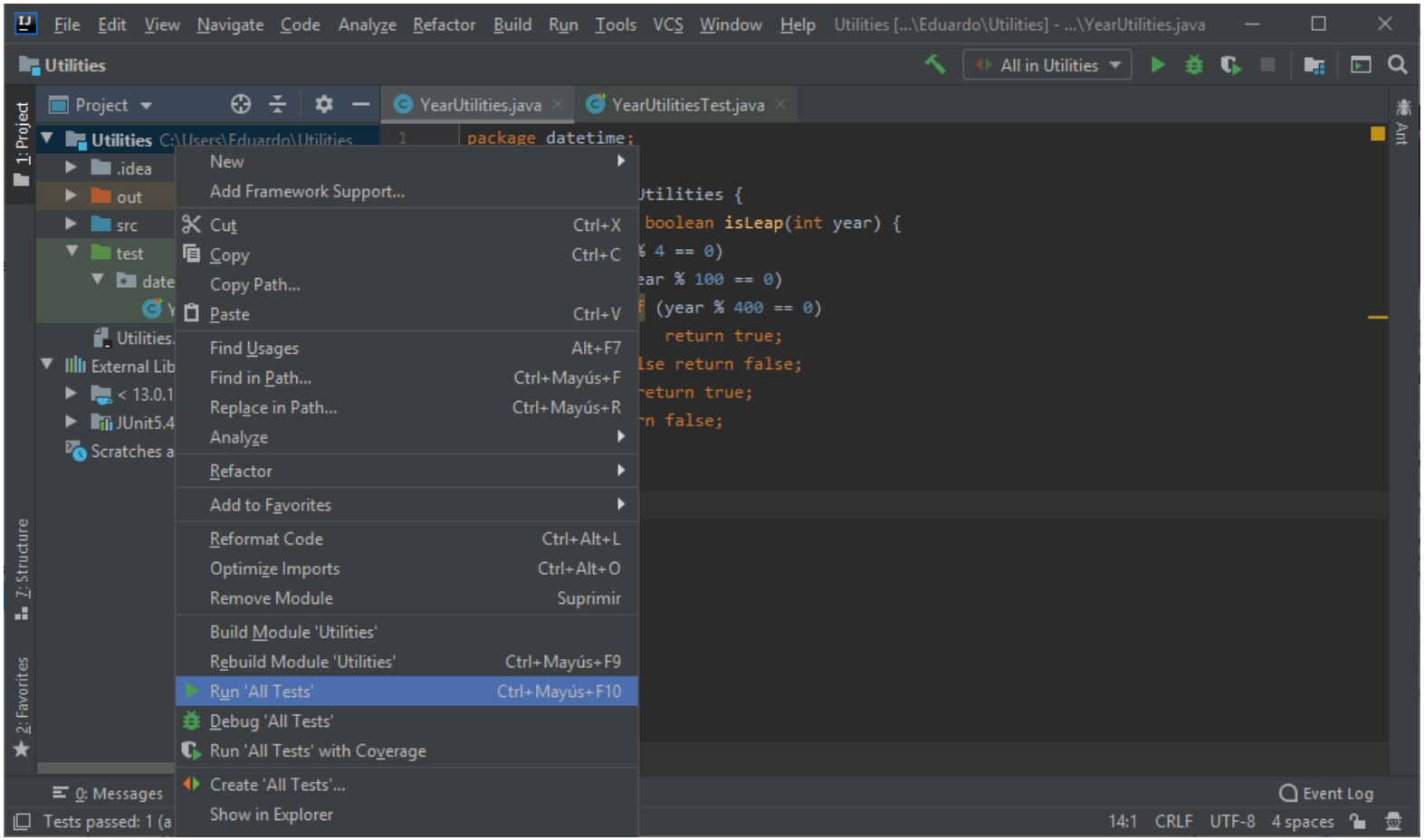
- To run the tests again, you can run once more the `YearUtilitiesTest` configuration (`Ctrl+F5`).
- If it passes the tests, the results are highlighted in green.

```
package datetime;

public class YearUtilities {
    public static boolean isLeap(int year) {
        if (year % 4 == 0)
            if (year % 100 == 0)
                if (year % 400 == 0)
                    return true;
                else return false;
            else return true;
        else return false;
    }
}
```

```
"C:\Program Files\Java\jdk-13.0.1\bin\java.exe" ...

Process finished with exit code 0
```

# Example

- You can also run all the tests in the project jointly from the *Run 'All Tests'* option that appears when you right-click on the project.



# Example

- If we make a mistake in the code and state that the years that are divisible by 400 are not leap years...

```
package datetime;

public class YearUtilities {
    public static boolean isLeap(int year) {
        if (year % 4 == 0)
            if (year % 100 == 0)
                if (year % 400 == 0)
                    return false;
                else return false;
            else return true;
        else return false;
    }
}
```

# Example

- ...you'll see an error in the tests for the one about the year 2000.

```
import static org.junit.jupiter.api.Assertions.*;

class YearUtilitiesTest {

    @Test
    void isLeap() {
        assertFalse(YearUtilities.isLeap( year: 2019)); // not leap, not divisible by 4
        assertTrue(YearUtilities.isLeap( year: 2020)); // leap, divisible by 4
        assertFalse(YearUtilities.isLeap( year: 1900)); // not leap, divisible by 4 and 100
        assertTrue(YearUtilities.isLeap( year: 2000)); // leap, divisible by 4, 100 and 400
    }
}
```

```
Tests failed: 1 of 1 test – 15 ms
<default package> 15 ms
YearUtilitiesTest 15 ms
isLeap() 15 ms
at datetime.YearUtilitiesTest.isLeap(YearUtilitiesTest.java:16) <19 internal calls>
at java.base/java.util.ArrayList.forEach(ArrayList.java:1507) <9 internal calls>
at java.base/java.util.ArrayList.forEach(ArrayList.java:1507) <18 internal calls>
at com.intellij.rt.junit.IdeaTestRunner$Repeater.startRunnerWithArgs(IdeaTestRunner.java:33)
at com.intellij.rt.junit.JUnitStarter.prepareStreamsAndStart(JUnitStarter.java:230)
```

# Example

- If we remove the test about the year 2000, the code will look like it's correct, but it isn't ⇒ Passing the tests does not prove that there aren't any errors, it depends on the quality of the tests.

```
import static org.junit.jupiter.api.Assertions.*;

class YearUtilitiesTest {

    @Test
    void isLeap() {
        assertFalse(YearUtilities.isLeap( year: 2019)); // not leap, not divisible by 4
        assertTrue(YearUtilities.isLeap( year: 2020)); // leap, divisible by 4
        assertFalse(YearUtilities.isLeap( year: 1900)); // not leap, divisible by 4 and 100
//      assertTrue(YearUtilities.isLeap(2000)); // leap, divisible by 4, 100 and 400
    }
}
```

```
Tests passed: 1 of 1 test – 12 ms
Test Results 12 ms
  YearUtilitiesTest 12 ms
    isLeap() 12 ms

"C:\Program Files\Java\jdk-13.0.1\bin\java.exe" ...

Process finished with exit code 0
```

# Example

- If we presuppose that there is no such thing as a year zero in the current calendar, we can throw an exception if someone passes the year zero as an argument.

```
package datetime;

public class YearUtilities {
    public static boolean isLeap(int year) {
        if (year == 0)
            throw new IllegalArgumentException();
        if (year % 4 == 0)
            if (year % 100 == 0)
                if (year % 400 == 0)
                    return true;
                else return false;
            else return true;
        else return false;
    }
}
```

# Example

- To check that a test actually throws an exception, we need to use the `assertThrows(Class<T> expectedType, Executable executable)` method.
- **First argument:**
  - The first argument is the class of the exception, and the easiest way to refer to it is to add the `.class` suffix to the class name.
  - For example: `IllegalArgumentException.class`
- **Second argument:**
  - The second argument is an `Executable` object, which is a functional interface that can be used to implement any block of generic code that potentially throws a `Throwable` object.
  - The easiest way to create a functional interface is to use a *lambda* expression with no parameters which executes the desired method.
  - Por ejemplo: `() -> YearUtilities.isLeap(0)`

# Example

- We create in our tests a new test method that checks that passing zero as an argument throws `IllegalArgumentException`.

```
@Test
void isLeap() {
    assertFalse(YearUtilities.isLeap( year: 2019)); // not leap, not divisible by 4
    assertTrue(YearUtilities.isLeap( year: 2020)); // leap, divisible by 4
    assertFalse(YearUtilities.isLeap( year: 1900)); // not leap, divisible by 4 and 100
    assertTrue(YearUtilities.isLeap( year: 2000)); // leap, divisible by 4, 100 and 400
}

@Test
void isZeroLeap() {
    assertThrows(IllegalArgumentException.class, () -> YearUtilities.isLeap( year: 0));
}
}
```

Run: YearUtilitiesTest.isZeroLeap

Tests passed: 1 of 1 test – 16 ms

```
"C:\Program Files\Java\jdk-13.0.1\bin\java.exe" ...

Process finished with exit code 0
```

# Example

- If at some point we don't want a specific test to run (e,g., because there is no code yet that passes it yet), we can omit it with the `@Disabled` annotation.

```
import org.junit.jupiter.api.Disabled;
import org.junit.jupiter.api.Test;

import static org.junit.jupiter.api.Assertions.*;

class YearUtilitiesTest {

    @Test
    void isLeap() {
        assertFalse(YearUtilities.isLeap( year: 2019)); // not leap, not divisible by 4
        assertTrue(YearUtilities.isLeap( year: 2020)); // leap, divisible by 4
        assertFalse(YearUtilities.isLeap( year: 1900)); // not leap, divisible by 4 and 100
        assertTrue(YearUtilities.isLeap( year: 2000)); // leap, divisible by 4, 100 and 400
    }

    @Test @Disabled
    void isZeroLeap() {
        assertThrows(IllegalArgumentException.class, () -> YearUtilities.isLeap( year: 0));
    }
}
```

Tests failed: 1, passed: 0, ignored: 1 (3 minutes ago)

# Example

- When running the tests, the disabled ones will be shown (so we don't forget about them) but highlighted in gray (not green or red).

```
@Test
void isLeap() {
    assertFalse(YearUtilities.isLeap( year: 2019)); // not leap, not divisible by 4
    assertTrue(YearUtilities.isLeap( year: 2020)); // leap, divisible by 4
    assertFalse(YearUtilities.isLeap( year: 1900)); // not leap, divisible by 4 and 100
    assertTrue(YearUtilities.isLeap( year: 2000)); // leap, divisible by 4, 100 and 400
}

@Test @Disabled
void isZeroLeap() {
    assertThrows(IllegalArgumentException.class, () -> YearUtilities.isLeap( year: 0));
}
}
```

```
Tests passed: 1, ignored: 1 of 2 tests – 15 ms
Test Results 15 ms
  YearUtilitiesTest 15 ms
    isLeap() 15 ms
    isZeroLeap()

void datetime.YearUtilitiesTest.isZeroLeap() is @Disabled
```

# Best practices

- **Do I have to write a test for everything?**
  - No, just test anything that could reasonably break.
  - Writing tests takes time. It's better to focus on detecting errors, rather than testing code that is "*too simple to break*".
- **What if simple things become complex in the future?**
  - It is true that adding tests for even these simple methods guards against the possibility that someone refactors and makes the methods "not-so-simple" anymore.
  - In that case, though, the refactorer needs to be aware that the method is now complex enough to break, and should write tests for it – and preferably before the refactoring.

# Best practices

- **What is "*too simple to break*"?**
  - The general philosophy is this: if it can't break on its own, it's too simple to break[3].
  - Inside this category we have trivial *getters* and *setters*.
  - Or methods that delegate their functioning in other methods.

## Too simple to break

```
public getValue() { return value; }

public setValue() { this.value = value; }

public void myMethod(final int a, final String b) {
    myCollaborator.anotherMethod(a, b);
}
```

[3] https://junit.org/junit4/faq.html#best_3

# Test execution order

- Well-written test code should not assume any order, i.e., tests should not depend on other tests.
- To ensure a certain state when executing a test we can use annotations like `@BeforeEach` and `@BeforeAll`.
- In JUnit 5 it is possible to choose the order of the tests by using the `@TestMethodOrder` and `@Order(number)` annotations.
- `@TestMethodOrder` is an annotation that is placed above the class. The common thing is to use `OrderAnnotation`, which represents a numeric order from greater to lesser.
- `@Order(number)` is an annotation above each test that indicates the running order number.
- More info at: `https://junit.org/junit5/docs/current/user-guide/#writing-tests-test-execution-order`

# Example

- If one of the new annotations has not been *imported*, use *Alt+Enter* to add new imports.

```
import org.junit.jupiter.api.Test;

org.junit.jupiter.api.MethodOrderer.OrderAnnotation? Alt+Intro
@TestMethodOrder(OrderAnnotation.class)
class YearUtilitiesTest {

    @Test
    void isLeap() {
        assertFalse(YearUtilities.isLeap( year: 2019)); // not leap, not divisible by 4
        assertTrue(YearUtilities.isLeap( year: 2020)); // leap, divisible by 4
        assertFalse(YearUtilities.isLeap( year: 1900)); // not leap, divisible by 4 and 100
        assertTrue(YearUtilities.isLeap( year: 2000)); // leap, divisible by 4, 100 and 400
    }

    @Test @Disabled
    void isZeroLeap() {
        assertThrows(IllegalArgumentException.class, () -> YearUtilities.isLeap( year: 0));
    }
}
```

Cannot resolve symbol 'OrderAnnotation'

# Example

- Example of running tests in a specific order.

```
@TestMethodOrder(MethodOrderer.OrderAnnotation.class)
class YearUtilitiesTest {
    @Test @Order(1)
    void isLeap() {
        assertFalse(YearUtilities.isLeap( year: 2019)); // not leap, not divisible by 4
        assertTrue(YearUtilities.isLeap( year: 2020)); // leap, divisible by 4
        assertFalse(YearUtilities.isLeap( year: 1900)); // not leap, divisible by 4 and 100
        assertTrue(YearUtilities.isLeap( year: 2000)); // leap, divisible by 4, 100 and 400
    }
    @Test @Order(2)
    void isZeroLeap() {
        assertThrows(IllegalArgumentException.class, () -> YearUtilities.isLeap( year: 0));
    }
}
```

Tests passed: 2 of 2 tests – 15 ms

# Example

- Same example as before but in a different order.

```
@TestMethodOrder(MethodOrderer.OrderAnnotation.class)
class YearUtilitiesTest {
    @Test @Order(2)
    void isLeap() {
        assertFalse(YearUtilities.isLeap( year: 2019)); // not leap, not divisible by 4
        assertTrue(YearUtilities.isLeap( year: 2020)); // leap, divisible by 4
        assertFalse(YearUtilities.isLeap( year: 1900)); // not leap, divisible by 4 and 100
        assertTrue(YearUtilities.isLeap( year: 2000)); // leap, divisible by 4, 100 and 400
    }
    @Test @Order(1)
    void isZeroLeap() {
        assertThrows(IllegalArgumentException.class, () -> YearUtilities.isLeap( year: 0));
    }
}
```

Run: YearUtilitiesTest — Tests passed: 2 of 2 tests – 16 ms

- Test Results 16 ms
  - YearUtilitiesTest 16 ms
    - isZeroLeap() 16 ms
    - isLeap()

"C:\Program Files\Java\jdk-13.0.1\bin\java.exe" ...

Tests passed: 2 (moments ago)

# Summary of the transition between JUnit 4 and JUnit 5

- **Imports**[4]. JUnit 5 uses the new `org.junit.jupiter.api` package for annotations and classes.
- **Annotations**. In JUnit 5, `@Test` loses its parameters (`timeout` is now the `@Timeout` annotation) and we get the new annotations `@BeforeEach` and `AfterEach` (formerly `@Before` and `@After`), `@BeforeAll` and `@AfterAll` (formerly `@BeforeClass` and `@AfterClass`) and `@Disabled` (formerly `@Ignore`).
- **Assertions**. The `Assertions` class belongs now to the `org.junit.jupiter.api` package, and the message is now the last parameter of the method. There is also a new method for `assertTimeout()`.
- **Lambda-Expressions**. They are used as arguments for methods like `assertThrows()`, which replaces the parameter `expected` that was previously a part of the `@Test` annotation.

[4] More info at `https://blogs.oracle.com/javamagazine/migrating-from-junit-4-to-junit-5-important-differences-and-benefits`

# Table of Contents

# Suitability of the tests

## Characteristics of the tests ⇒ Quantity and representativeness

- **Quantity**: A high number of tests is desirable.
- **Representativeness**: Tests must be representative of the whole range of possible situations.

## How can we measure representativeness? ⇒ Code coverage

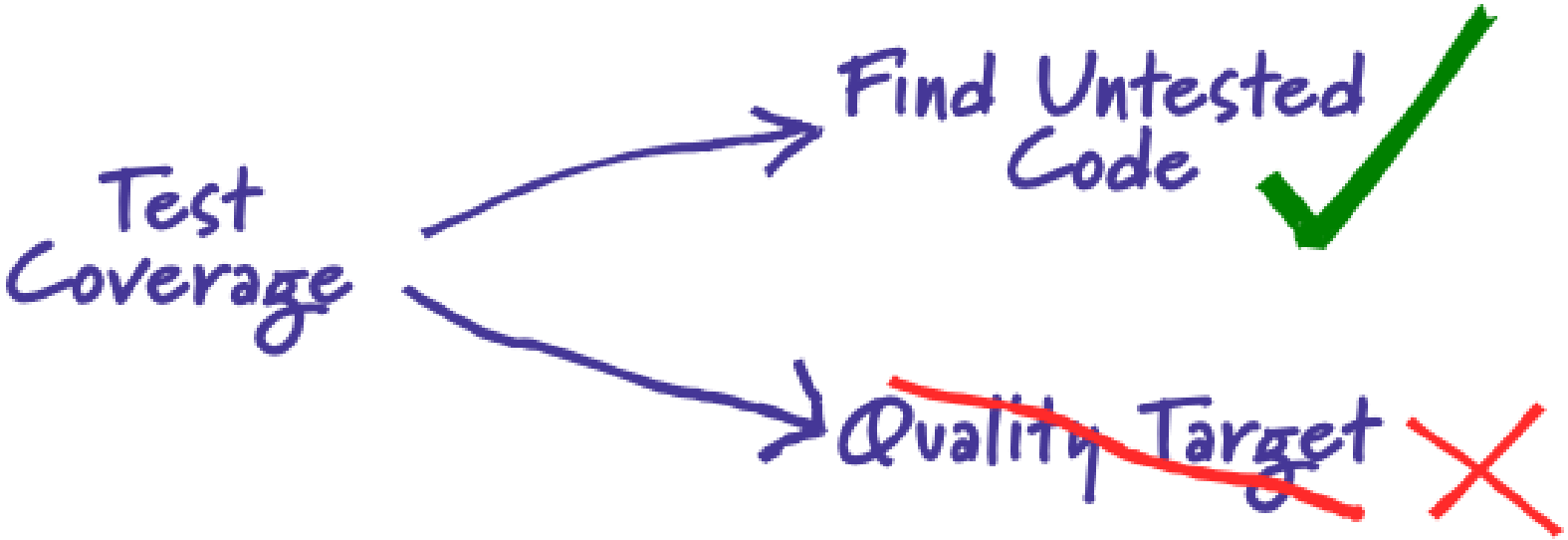
- **Line coverage**: Number of lines traversed by the tests.
- **Branch coverage**: Number of branches in conditional statements that are traversed by the tests.

# Code coverage

## Code coverage objective

To find untested parts of a codebase.

- **Test coverage is of little use as a numeric statement of how good your tests are.**[5]




[5] https://martinfowler.com/bliki/TestCoverage.html

# Code coverage

- **Why is test coverage of little use as a numeric statement of how good your tests are?**
  - High coverage numbers are too easy to reach with low quality testing techniques such as "Assertion Free Testing"[6]: tests without assertions that go through the code without testing anything. They only test that unexpected exceptions such as `NullPointerException` are not being thrown.
  - It is easy to obtain a high code coverage testing "*too simple to break*" methods while leaving fundamental parts of your code without tests.

**Important!**

Low coverage numbers are a sign of trouble in tests. But high numbers don't necessarily mean absence of problems in tests.

[6] https://martinfowler.com/bliki/AssertionFreeTesting.html

# Example

- If we remove part of the code of our tests, we can see that there are parts of the source code that are not being tested.

```
class YearUtilitiesTest {
    @Test
    void isLeap() {
        assertFalse(YearUtilities.isLeap( year: 2019)); // not leap, not divisible by 4
        assertTrue(YearUtilities.isLeap( year: 2020)); // leap, divisible by 4
        assertFalse(YearUtilities.isLeap( year: 1900)); // not leap, divisible by 4 and 100
//          assertTrue(YearUtilities.isLeap(2000)); // leap, divisible by 4, 100 and 400
    }
//  @Test
//  void isZeroLeap() {
//      assertThrows(IllegalArgumentException.class, () -> YearUtilities.isLeap(0));
//  }
}
```

Run: YearUtilitiesTest — Tests passed: 2 of 2 tests – 16 ms

- Test Results 16 ms
  - YearUtilitiesTest 16 ms
    - isZeroLeap() 16 ms
    - isLeap()

```
"C:\Program Files\Java\jdk-13.0.1\bin\java.exe" ...
```

Tests passed: 2 (3 minutes ago)

# Example

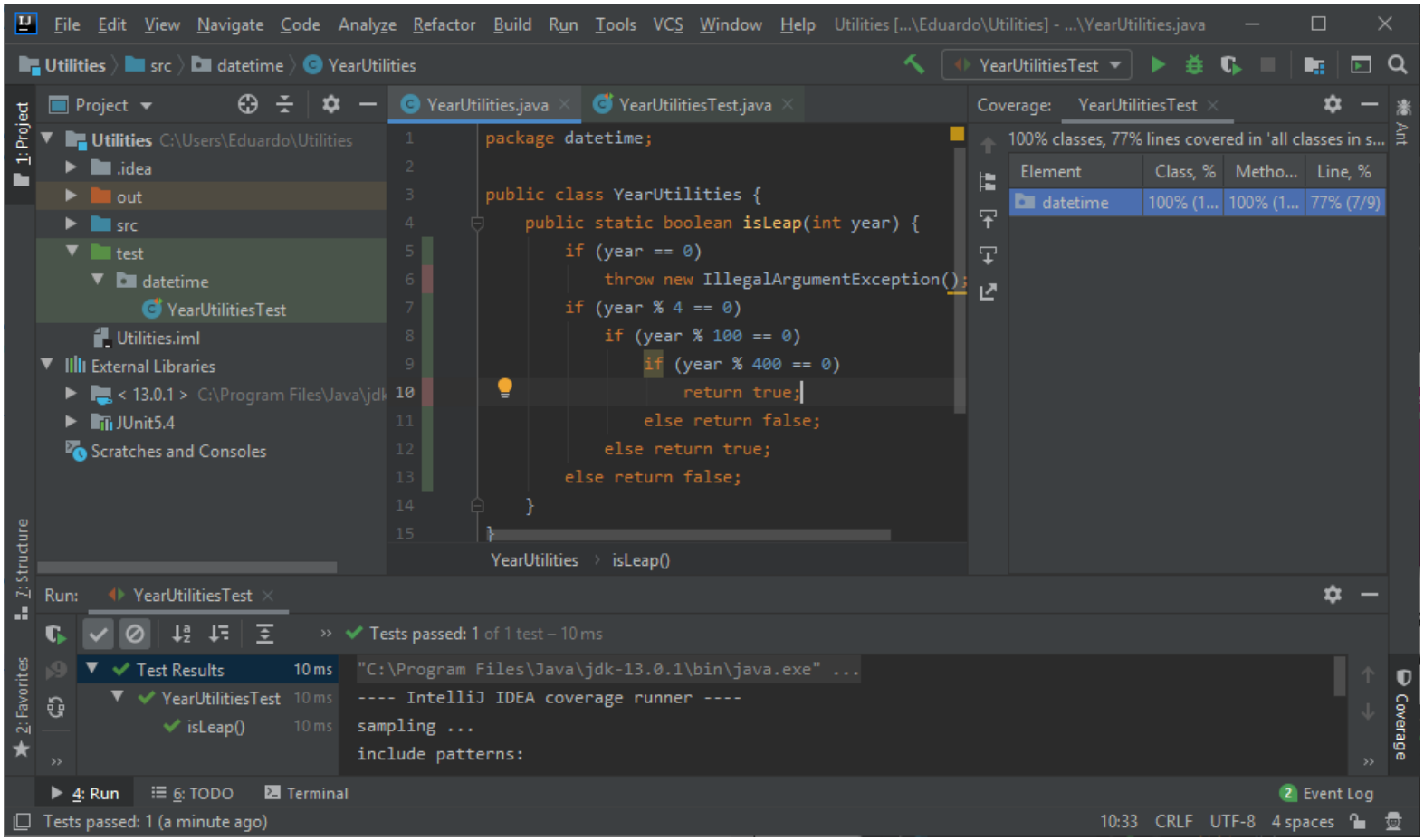
- To see our tests' coverage you simply have to run them using the option *Run ... with Coverage*

```
class YearUtilitiesTest {
    @Test
    void isLeap() {
        assertFalse(YearUtilities.isLeap( year: 2019)); // not leap, not divisible by 4
        assertTrue(YearUtilities.isLeap( year: 2020)); // leap, divisible by 4
        assertFalse(YearUtilities.isLeap( year: 1900)); // not leap, divisible by 4 and 100
//        assertTrue(YearUtilities.isLeap(2000)); // leap, divisible by 4, 100 and 400
    }
//    @Test
//    void isZeroLeap() {
//        assertThrows(IllegalArgumentException.class, () -> YearUtilities.isLeap(0));
//    }
}
```

Run selected configuration with coverage enabled

# Example

- We can see that there are tests for all classes (one) and all methods (one) but only (77 %) of the lines have been covered. Two lines marked in red are being neglected.

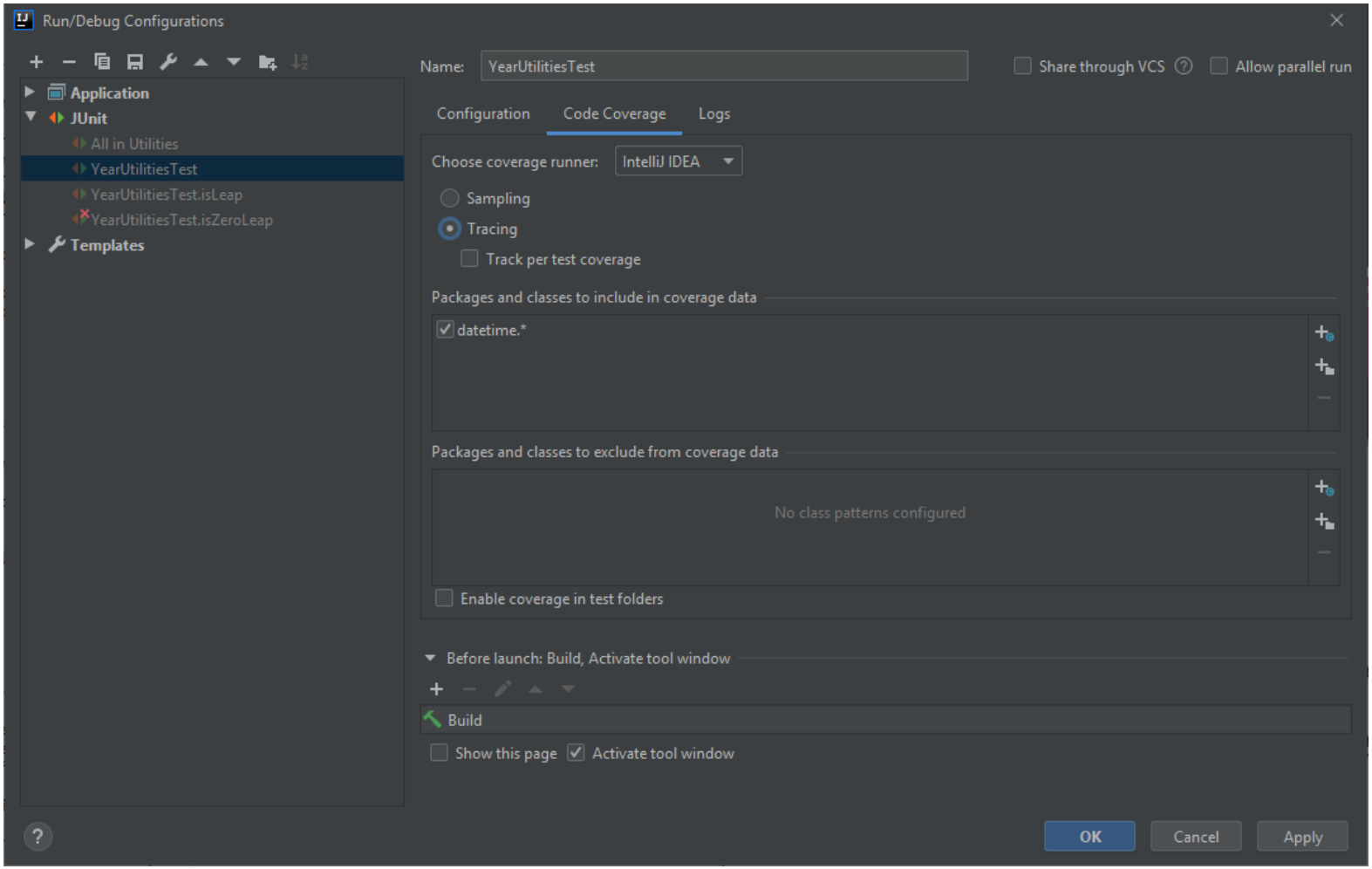


# Example

- In order to see branch coverage, you must enable the *Tracing* option.
- From the main menu, we select *Run — Edit Configurations*
- On the JUnit section, we go to the configuration of our test `YearUtilitesTest` and click on the *Code Coverage* tab.
- There are three options for coverage:
  - **Sampling**: It's run by default. Includes only line coverage and is the quickest one.
  - **Tracing**: Includes branch coverage.
  - **Track per test coverage**: It's included in the previous one. Allows tracking individual coverage for each test case.

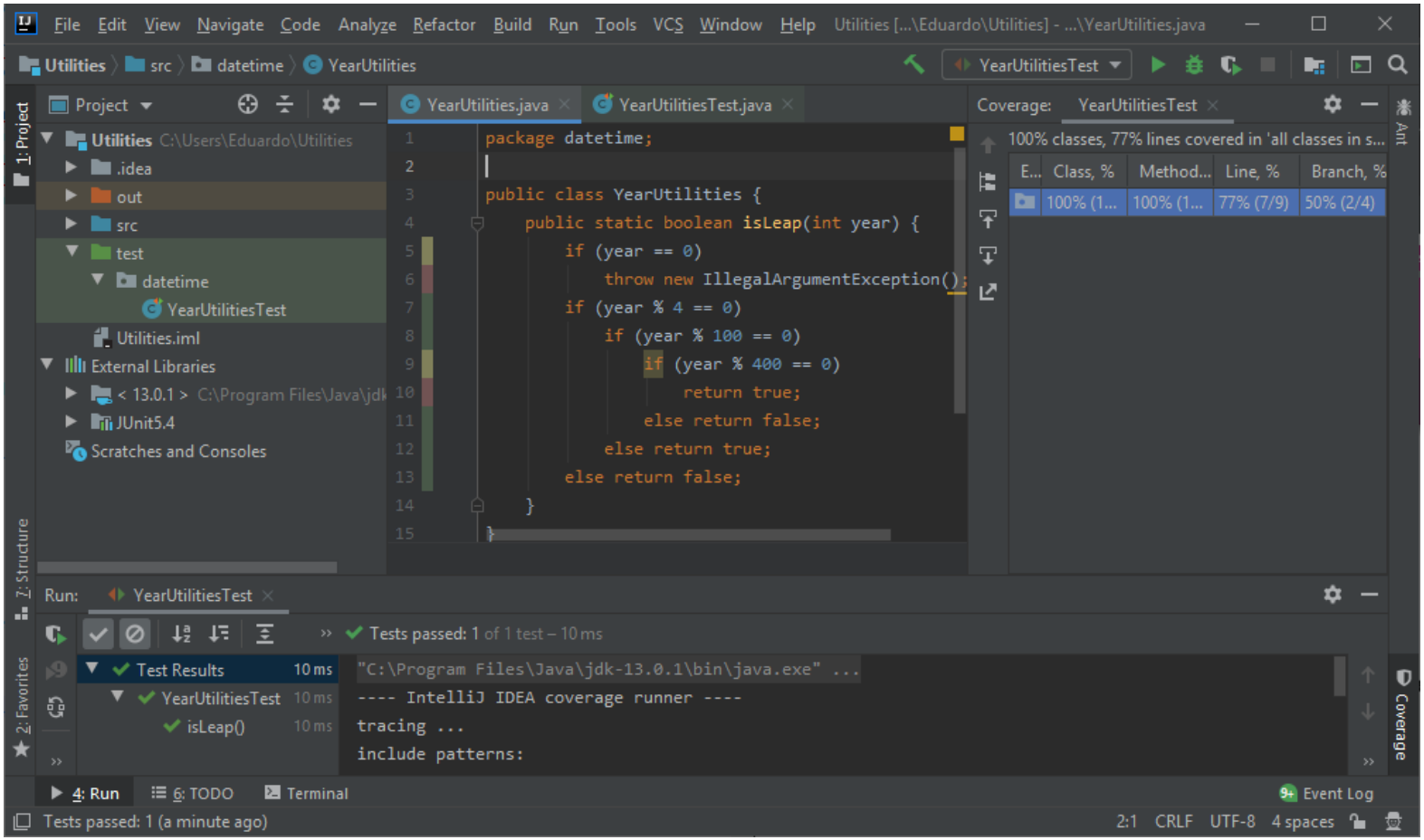
# Example

- Configuring test execution to include branch coverage.



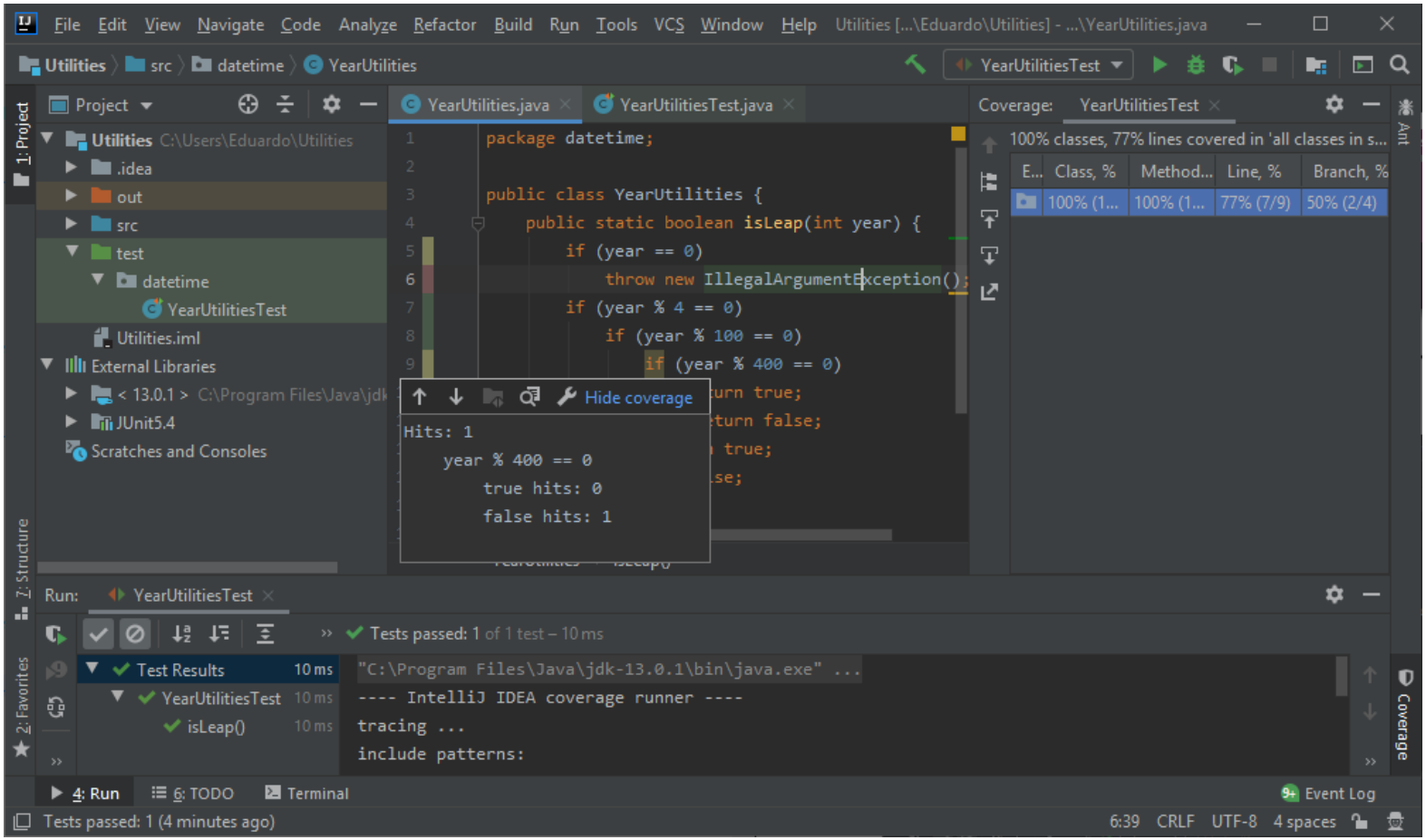
# Example

- When we run the tests again with coverage, it says now that branch coverage is only 50 %.



# Example

- Clicking on the color that represents coverage, we can see how many *hits* has has every branch. In the example we can see that the `true` branch for the 400 is never visited.



# Practice 3: JUnit

## Software Design (614G01015)

**Eduardo Mosqueira Rey** (Coordinador)

Department of Computer Science and Information Technology
Faculty of Computer Science

UNIVERSIDADE DA CORUÑA