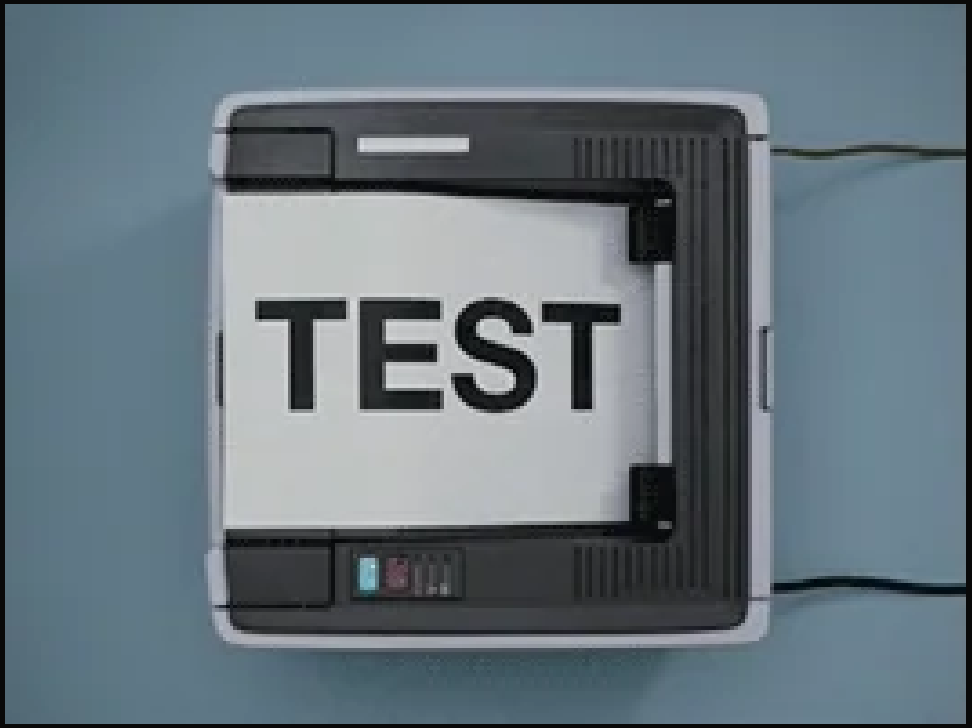
SOFTWARE TESTS



QUALITY ASSURANCE

Let's discuss 🔊

- How can quality be assured within a software project?

COVERED TOPICS

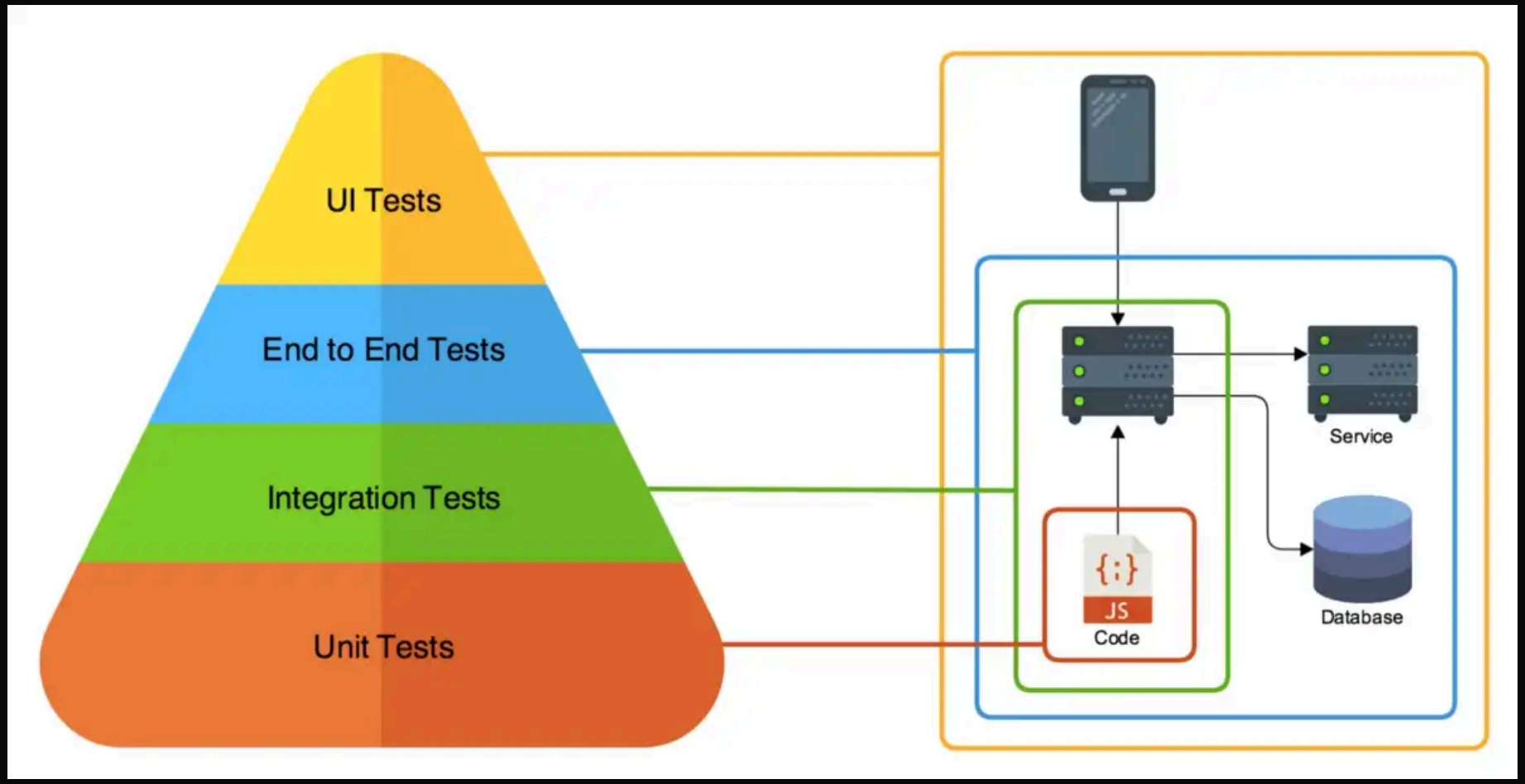
- Theory of testing
- Unit Tests
- TDD (Test Driven Development)
- Mocking Frameworks

THEORY OF TESTING

Running a program in a controlled way to find bugs

TEST LEVELS (ISTQB)

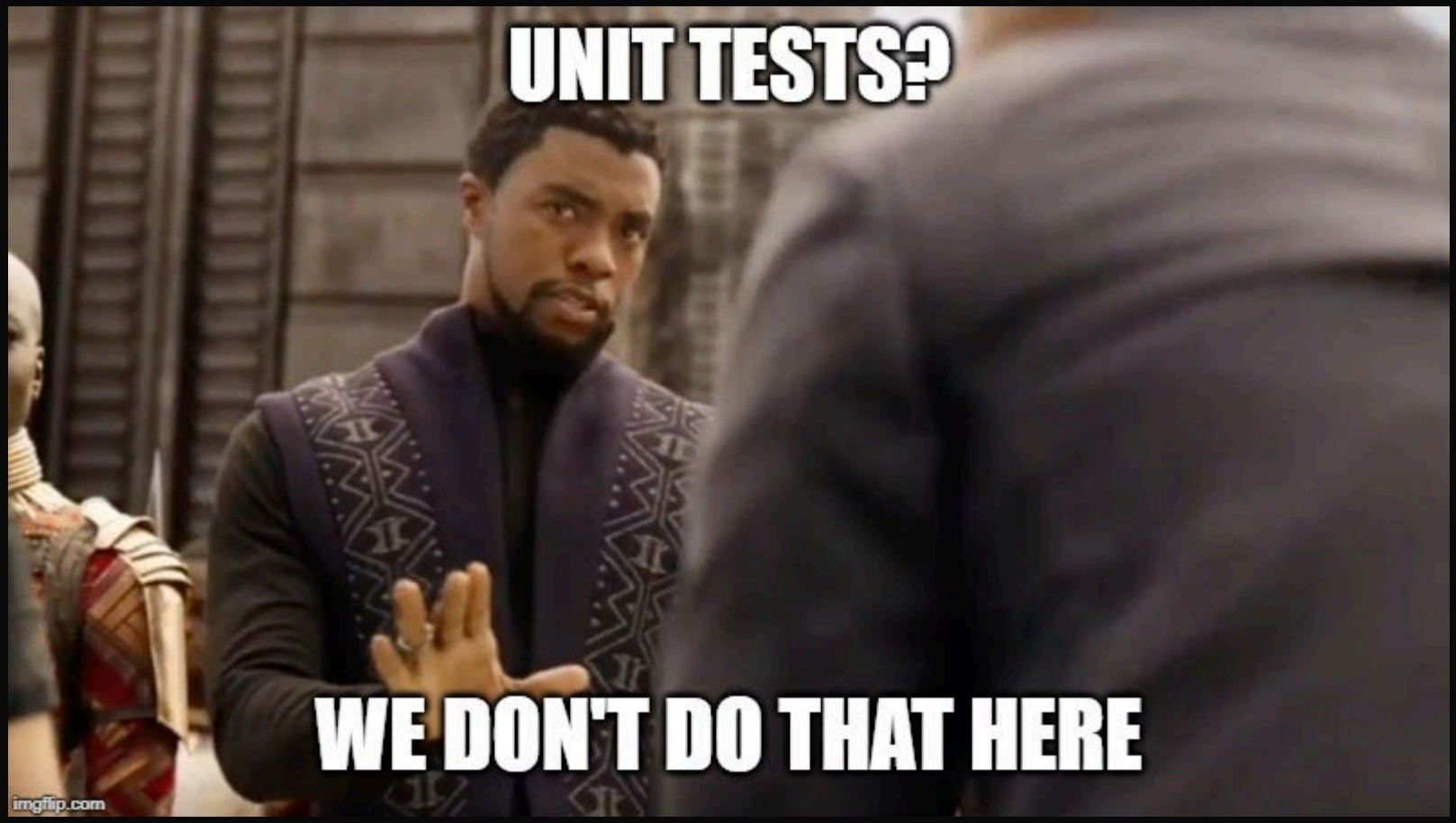
1. Unit tests
2. Integration tests
3. System tests (end to end)
4. Acceptance tests (UI)



Source: https://miro.medium.com/max/1400/1*S-WQ9KwM7kkmwKWy41SPYw.webp

UNIT TESTS

- Software, which checks my software
- testing single methods and classes (=smallest possible **units**)
- in Java: using the JUnit Framework (various versions available)

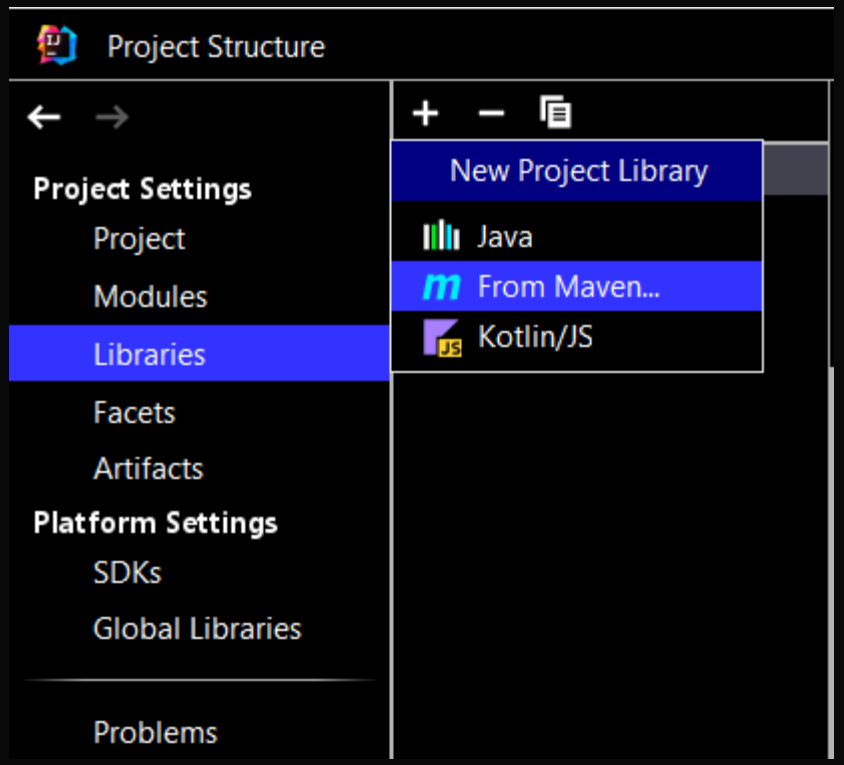

UNIT TESTS?
WE DON'T DO THAT HERE
imgflip.com

HANDS ON 👈

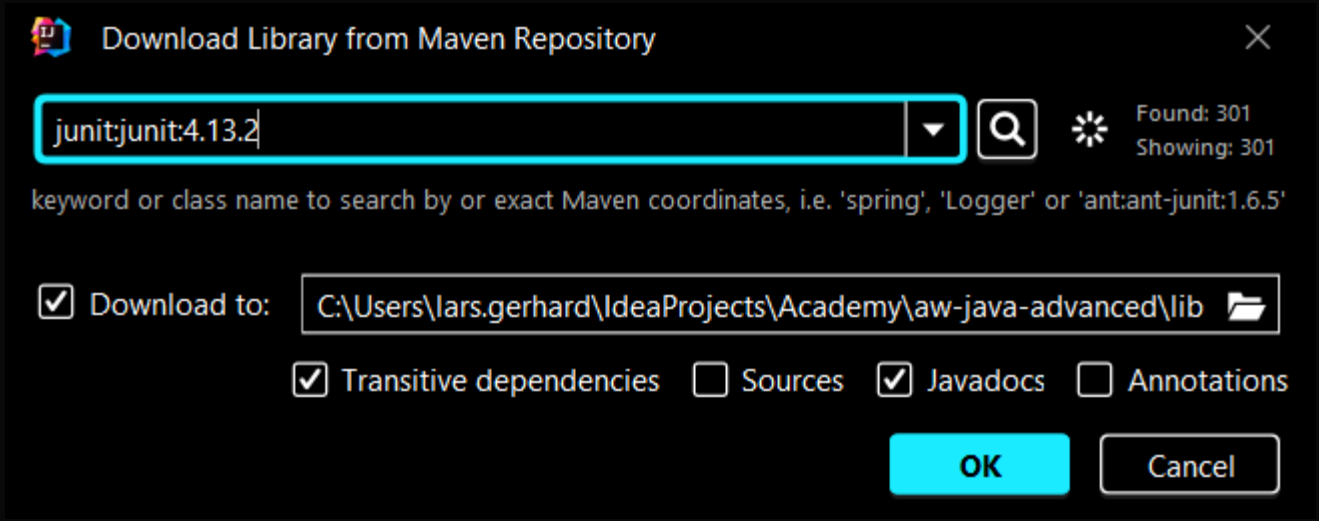
- Open the calulator project (prestudies, week 4)

```
public class Calculator {
    private int result;
    public Calculator() { this.result = 0; }
    public int getResult() { return result;  }
    public void plus(int value) { result += value; }
    public void minus(int value) { result -= value; }
    public void times(int value)  { result *= value; }
    public void divided(int value)  { result /= value; }
}
```

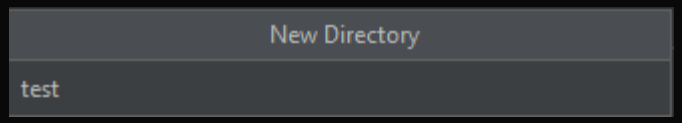
SETUP JUNIT IN INTELLIJ



SETUP JUNIT IN INTELLIJ



GENERATE A NEW TEST DIRECTORY



MARK DIRECTORY AS TEST SOURCE

Project

untitled C:\Users\lars.gerhard\IdeaProjects\Academy\delete-testing\untitled

- .idea
- lib
- src
  - Calculator
- test
- untitl
- External
- Scratche

Calculator.java

```
no usag
1  publi
6
2  p
3
n
4  p
7
n
8  p
11
12 p
15
n
16 p
19
n
20 p
23
n
24 p
27 }
28
```

- New
- Cut Ctrl+X
- Copy Ctrl+C
- Copy Path/Reference...
- Paste Ctrl+V
- Find Usages Ctrl+G
- Find in Files... Ctrl+H
- Replace in Files...
- Analyze
- Refactor
- Bookmarks
- Reformat Code Ctrl+Alt+L
- Optimize Imports Ctrl+Alt+O
- Delete... Delete
- Build Module 'untitled'
- Rebuild Module 'untitled' Ctrl+Shift+F9
- Open In
- Local History
- Repair IDE on File
- Reload from Disk
- Compare With... Ctrl+D
- Mark Directory as
  - Sources Root
  - Test Sources Root
  - Resources Root
- Convert Java File to Kotlin File Ctrl+Alt+Shift+K

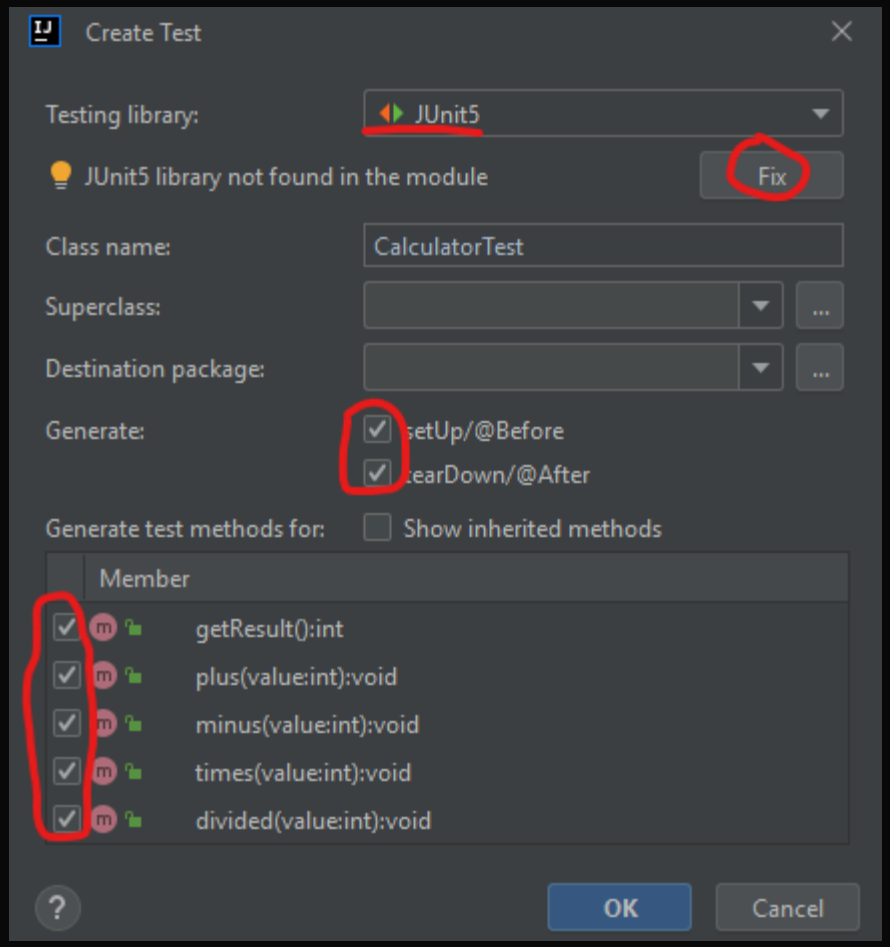
GENERATE A TEST CLASS

```
8    public int getResult() { return result; }
11
12   public void plus(int value) { result += value; }
15
     no usages
16   public void mi                    result -= value; }
19
     no usages
20   public void ti                    result *= value; }
23
     no usages
```

Generate

- Constructor
- Getter
- Setter
- Getter and Setter
- equals() and hashCode()
- toString()
- Override Methods...
- Test...
- Copyright

CHOOSE OPTIONS



- Click the "fix" button, to download JUnit 5

ADD YOUR FIRST TEST

```
@Test
void getResult() {
    Calculator c  = new Calculator();
    assertEquals(0, c.getResult(), "Result should be 0");
}
```

Let it run... 🏃

ADD YOUR FIRST TEST

```
@Test
void getResult() {
    Calculator c  = new Calculator();
    assertEquals(0, c.getResult(), "Result should be 0");
}
```

- Annotation marks a method as test method

Let it run... 🏃

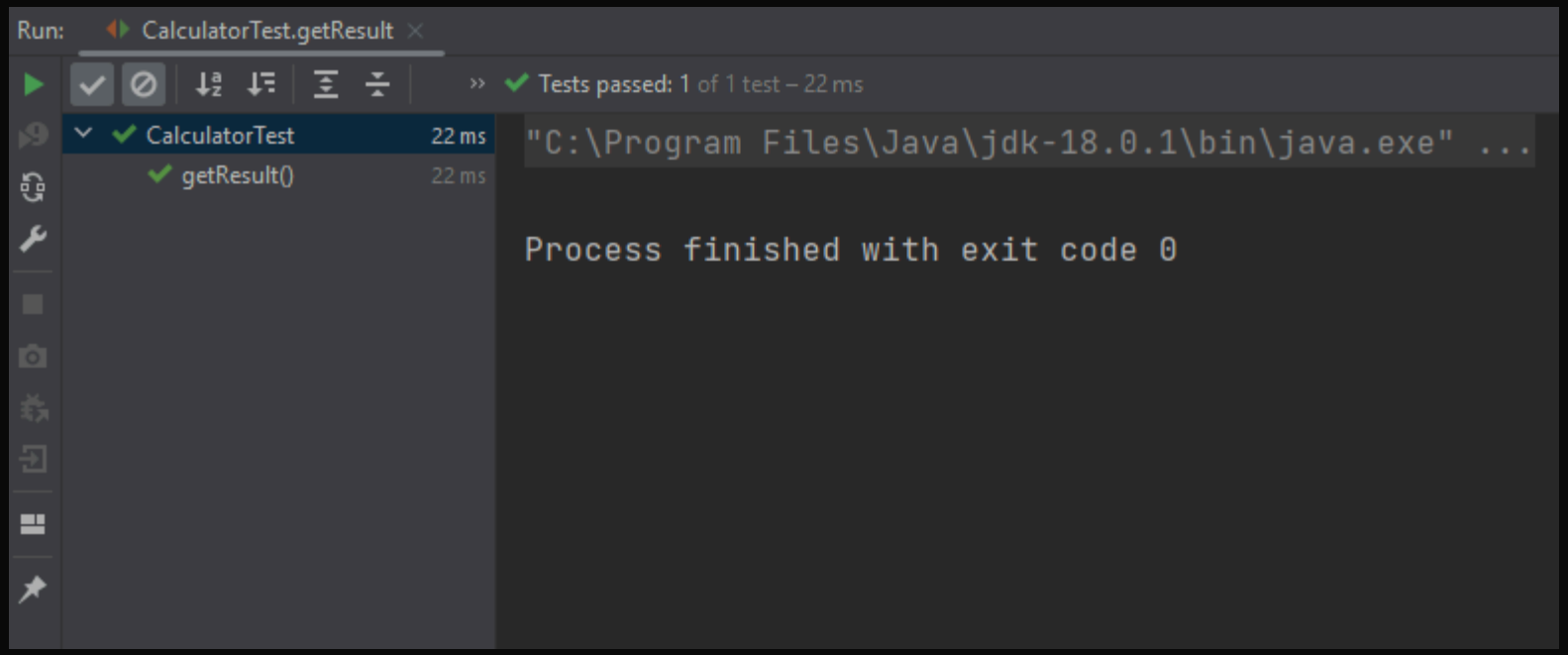
ADD YOUR FIRST TEST

```
@Test
void getResult() {
    Calculator c  = new Calculator();
    assertEquals(0, c.getResult(), "Result should be 0");
}
```

- Annotation marks a method as test method
- assert methods do the matching between an "expected" and the actual value

Let it run... 🏃

SUCCESS! 🙌



Checkout: "Run with coverage"

ASSERTION METHODS

- the "testing" moment
- (at least) two parameters:
  1. expected: the value your tested method should produce
  2. actual: the "actual" return value of your called method
- available for all primitive types + Objects
- available for arrays in JUnit 5: `assertArrayEquals()`

JUNIT 4 VS. JUNIT 5 ⚔️

- different annotations
- new types of assert methods
- non-public test methods and classes are allowed

https://howtodoinjava.com/junit5/junit-5-vs-junit-4/


JUNIT TEST
WORKING CODE
ifunny.co

👉 YOUR TURN!

Write tests and execute them right away:

1. primitive tests for all four basic calculations
2. A longer test, combining all calculation types
3. A test with numbers greater than 10000
4. A test with negative number as result
5. A test with negative number as parameter, e.g. plus(-4)

👉 YOUR TURN!

Now break the calculator on purpose: Make one method behave wrong, e.g. always +1 the result.

- How is the error noticeable?
- How can you re-run only the failed tests?

Bonus: Test with result greater than 3 billion, Test division by 0

👉 CHANGE THE IMPLEMENTATION

- Change the implementation of `times()` so it does not use the * operator, but a loop of additions.
- The result should still be correct and pass all tests. Make sure it also works for `0`, `1` and negative values.

Bonus: Write tests for `divided()`, where rounding of ints happens. Now replace the implementation of divided() with a loop of subtractions.

👉 CALCULATOR NEXT LEVEL 🚀

- New method `clear()`, which sets the result to 0
- Additional constructor with one parameter for the initial result.
- Method `absolute()`, which sets the result to its positive (absolute) value. (e.g. absolute(-5) is 5; absolute(3) is 3)

Don't change the existing tests - add new reasonable tests if needed!

👉 CALCULATOR NEXT LEVEL 🚀 - BONUS

- Method `power(int exponent)`: potentiates the result
- Method `round(int digits)`: rounds the result to a given number of significant digits

👀 REVIEW AND PREVIEW 🔭

- Why are tests useful?
- Where are tests stored in a project?
- What if I have a test for multiple classes?

👀 REVIEW AND PREVIEW 🔭

- Assert methods
- Which ones were useful so far?
- Order of arguments?

❓Implementation first - tests second❓

TEST DRIVEN DEVELOPMENT (TDD)

Source: https://methodpoet.com/wp-content/uploads/2022/02/tdd.png

TDD

TDD - THREE PHASES

1. Tests are written that initially fail.
2. Exactly as much (!) code is written as is necessary to pass the tests successfully.
3. The code and tests are refactored.

TDD - THREE PHASES

The three phases do not overlap! Each activity is assigned to one phase!

- tests are only written in the "write tests" phase.
- tests and code are only simplified in the 3rd phase ("Refactoring tests and code").
- no tests are written in phase 2 ("writing code").

TDD - AN INCREMENTAL APPROACH

Test-driven development describes incremental programming and is an evolutionary procedure.

- Each TDD cycle adds a new functionality to the software.
- The duration of a cycle is in the range of minutes to hours.

TDD HELLO WORLD

1. WRITE TEST

```
public class TDDTest {
    @Test
    public void testHelloService() {
        HelloService testObject = new HelloService();   // arrange
        String message = testObject.getMessage("Lars"); // act
        Assertions.assertEquals("Hello Lars", message); // assert
    }
}
```

Test fails ❌

2. IMPLEMENT AS MUCH AS NEEDED

```
public class HelloService {
    public String getMessage(String name) {
        return "Hello " + name;
    }
}
```

Test passes ✔

3. REFACTOR

not much to do 👼 just one little thing:

The `testObject` can be initialized in a `@BeforeEach` method (JUnit 5)

```
public class TDDTest {
  private HelloService testObject;

  @BeforeEach
  public void initTestEnvironment() {
    testObject = new HelloService();
  }

  @Test
  public void testHelloService() {
    String message = testObject.getMessage("Lars");
    Assertions.assertEquals("Hello Lars", message);
  }
}
```

NEXT TDD CYCLE

EAT.
SLEEP.
SLEEP.
REPEAT.

1. WRITE TEST

create a test that expects an exception for null values

```
@Test
public void testHelloException() {
    Assertions.assertThrows(Exception.class, () -> testObject.getM
}
```

1. WRITE TEST - JUNIT4 STYLE

```
@Test(expected = java.lang.Exception.class)
public void testHelloExceptionJUnit4() {
    String message = testObject.getMessage(null);
}
```

2. IMPLEMENT AS MUCH AS NEEDED

```
public String getMessage(String name) throws Exception {
    if (name == null) {
        throw new Exception("Input argument was null!");
    }
    return "Hello " + name;
}
```

3. REFACTOR

adjust the test from the first cycle

```
@Test
public void testHelloService() {
    String message = null;
    try {
        message = testObject.getMessage("Lars");
    } catch (Exception e) {
        throw new RuntimeException(e);
    }
    Assertions.assertEquals("Hello Lars", message);
}
```

TDD EXERCISE: CASH REGISTER 💵

Two classes:

- Product: name, price (in Cent as int)
- Register: scans products, calculates subtotal

TEMPLATE: PRODUCT

```
public class Product {
    public Product(String name, int price) { }

    public String getName() {
        return null;
    }

    public int getPrice() {
        return 0;
    }
}
```

TEMPLATE: REGISTER

```
public class Register {
   public void scan(Product product) { }

   public int getSubtotal() {
       return 0;
   }
}
```

CASH REGISTER 💵 PART 1

Perform TDD cycles for each step 👣

1. Product is constructed with name and price. Test examples: Correct value from `getName()`, `getPrice()`
2. Product is scanned by register. Test examples: Correct value from `getSubtotal()`
3. Multiple products scanned by register. Test examples: Correct values from `getSubtotal()` after every scan

CASH REGISTER 💵 PART 2

Perform TDD cycles for each step 👣

1. Add method `pay()` to cash register: Returns the amount to pay and resets it, so it's the next customer's turn. Test example: Correct amount returned, afterwards correct value from `getSubtotal()`
2. Add method `pay(int paidAmount)` : Like above, but returns change amount for the customer.

CASH REGISTER 💵 PART 3

Perform TDD cycles for each step 👣

1. Register handles credit (e.g. from returned bottle deposit)
2. Discount voucher: 10%, also applies to all the following customer products
3. Cash register can cancel the last scanned product immediately ("storno").

CASH REGISTER 💵 BONUS 1

Some products are part of a loyalty program. When purchasing loyalty products worth at least € 10 (in one purchase), the customer receives a 5% discount on the purchase. Also note special cases with credits for other discounts.

CASH REGISTER 💵 BONUS 2

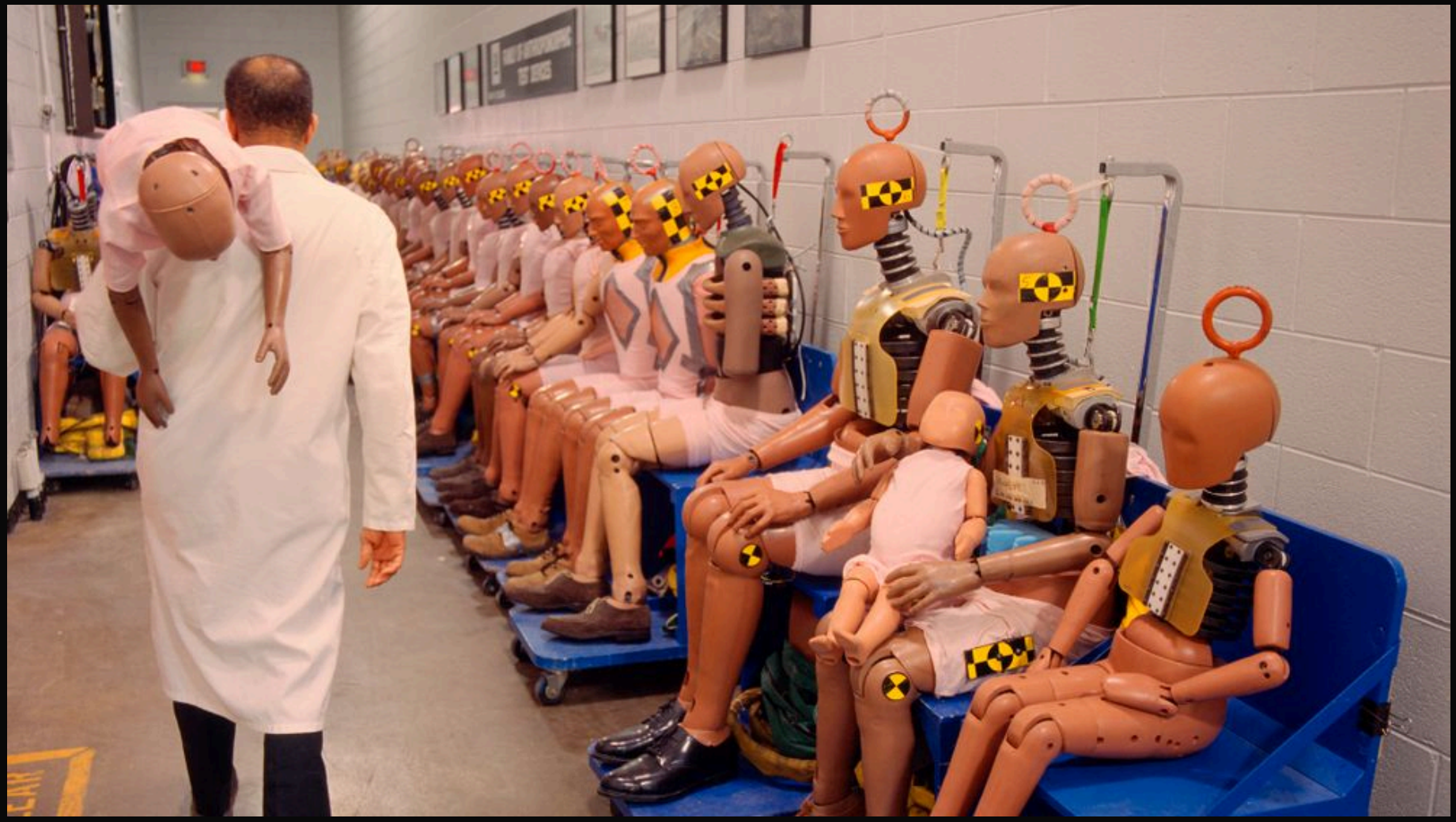
The management would like to know at the end of the day:

- How many customers have shopped
- How much turnover was made
- How much turnover was generated per purchase on average
- How much discount was granted through the 10% discount and loyalty program
- What percentage of customers shopped for more than € 100

MOCKING FRAMEWORKS

WHAT IS MOCKING?

- unit-tests shall cover exactly one functionality
- dependencies shall not be relevant (classes are either tested on their own or are trusted)
- test-object sometimes needs a "replacement" for dependencies - this is a **Mock**!



source: Spiegel Online

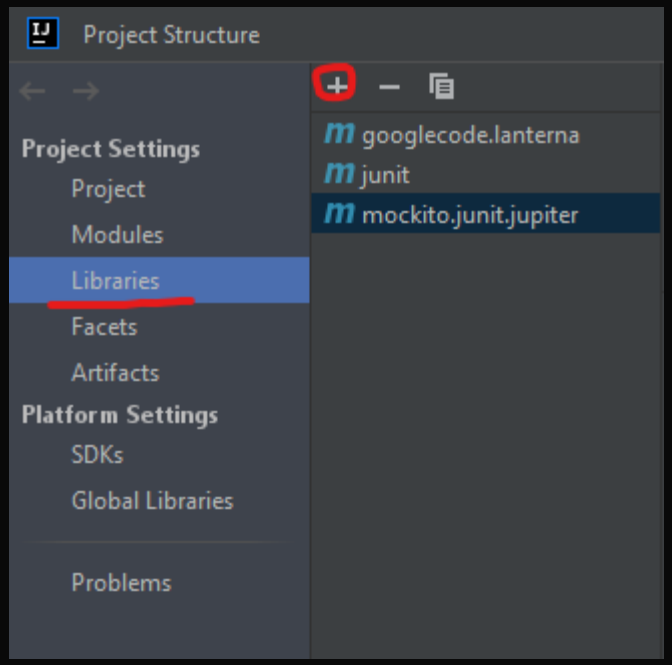
WHEN SHALL I USE MOCKS? 🤔

- not everything has to be mocked
- Integration tests can do the job as well!

SETTING UP MOCKITO 🍹

Add a maven library to your project:

- `org.mockito:mockito.junit.jupiter` for JUnit 5 (we will use that!)
- `org.mockito:mockito.core` for JUnit 4


Project Structure
Project Settings
Project
Modules
Libraries
Facets
Artifacts
Platform Settings
SDKs
Global Libraries
Problems
googlecode.lanterna
junit
mockito.junit.jupiter

EXAMPLE: SHOP 🛍️

- A shop is using a database to store its articles.
- We want to test the shop functionality without hosting a database
- Mocking is needed to do the test!

EXAMPLE: SHOP

```
public class Shop {
    private Database database;
    public Shop(Database database) {
        this.database = database;
    }
    public boolean query(String query) {
        return database.isAvailable();
    }
    @Override
    public String toString() {
        return "Using article-database with id: " + String.valueOf
    }
}
```

DATABASE DUMMY

```
public class Database {
    public boolean isAvailable() {
        // currently not implemented, as this is just demo used in
        return false;
    }
    public int getUniqueId() {
        return 42;
    }
}
```

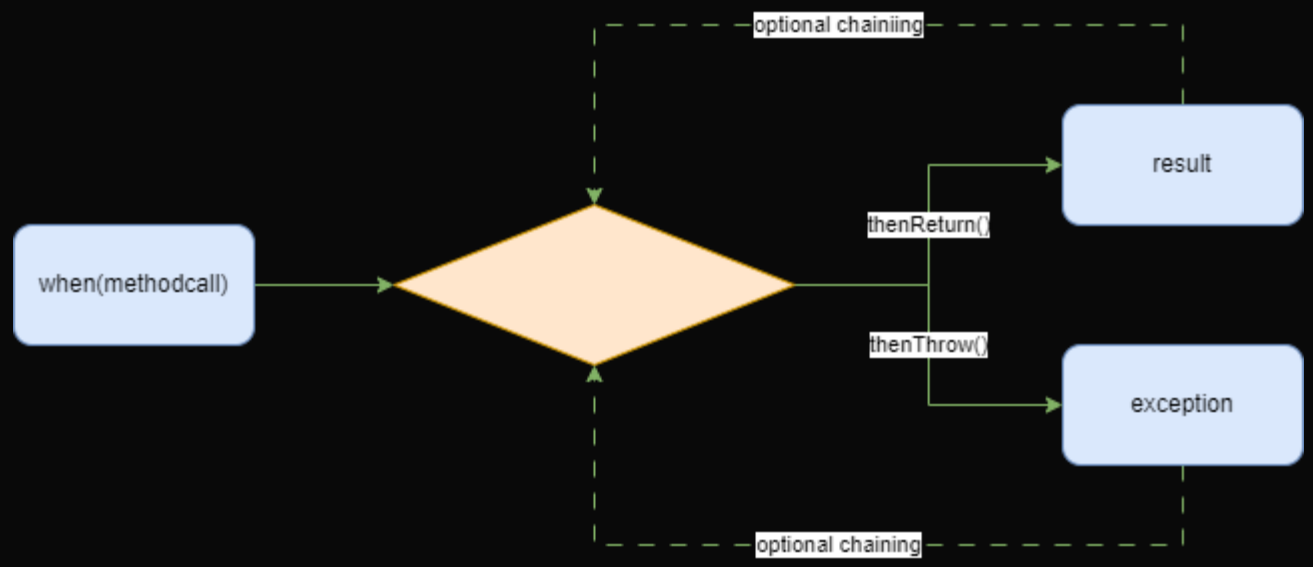
TEST CLASS

```
1  @ExtendWith(MockitoExtension.class)
2  class ShopTest {
3      @Mock
4      Database databaseMock;
5
6      @Test
7      public void testQuery()  {
8          assertNotNull(databaseMock);
9          when(databaseMock.isAvailable()).thenReturn(true);
10         Shop s = new Shop(databaseMock);
11         assertTrue(s.query("SELECT * from s"));
12     }
13 }
```

TEST CLASS

```
1  @ExtendWith(MockitoExtension.class)
2  class ShopTest {
3      @Mock
4      Database databaseMock;
5
6      @Test
7      public void testQuery()  {
8          assertNotNull(databaseMock);
9          when(databaseMock.isAvailable()).thenReturn(true);
10         Shop s = new Shop(databaseMock);
11         assertTrue(s.query("SELECT * from s"));
12     }
13 }
```

MOCKITO SYNTAX



- chained values are returned in the specified order until the last one is used.
- after all values are used the last one is used if the method is called again

MOCKITO SPY 🕵️

- use @Spy to wrap a real object

```
@Spy
List<String> articles = new LinkedList<>();
```

MOCKITO SPY 

- calls are delegated to that Spy-object

```
@Test
public void testArticleSpy() {
    // when(articles.get(0)).thenReturn("foo");
    // does not work because the delegate is called.
    // So articles.get(0) throws IndexOutOfBoundsException

    // you have to use doReturn() here
    doReturn("foo").when(articles).get(0);

    assertEquals("foo", articles.get(0));
}
```

BEHAVIOURAL TESTING

- check if a method is called with the right parameters
- don't check the results
- use `verify()` to do that

```java
@Test
public void testBehaviour(@Mock Database database){
        // create and configure mock
        when(database.getUniqueId()).thenReturn(43);

        // call method testing on the mock with parameter 12
        database.setUniqueId(12);
        database.getUniqueId();
        database.getUniqueId();

        // check if the setter was called with the parameter 12
        verify(database).setUniqueId(ArgumentMatchers.eq(12));

        ///...
}
```

```java
public void testBehaviour(@Mock Database database){
    //....
    // was the getter called twice?
    verify(database, times(2)).getUniqueId();

    // other alternatives for verifying the number of method calls
    verify(database, never()).isAvailable();
    verify(database, never()).setUniqueId(13);
    verify(database, atLeastOnce()).setUniqueId(12);
    verify(database, atLeast(2)).getUniqueId();

    // This checks that no other methods were called on this objec
    // You can call it after you have verified the expected method
    verifyNoMoreInteractions(database);
}
```

CASH REGISTER 💵 MEETS MOCKITO 🍹

Refactor the cash register code:

- Products are saved in a database (accessible by a unique ID `barcode`)
- The database is connected to the register
- Revise as many of your written test methods as possible
- mock the behaviour from before by using Mockito
- Bonus: use a `Spy` and the `verify()` method in a new test