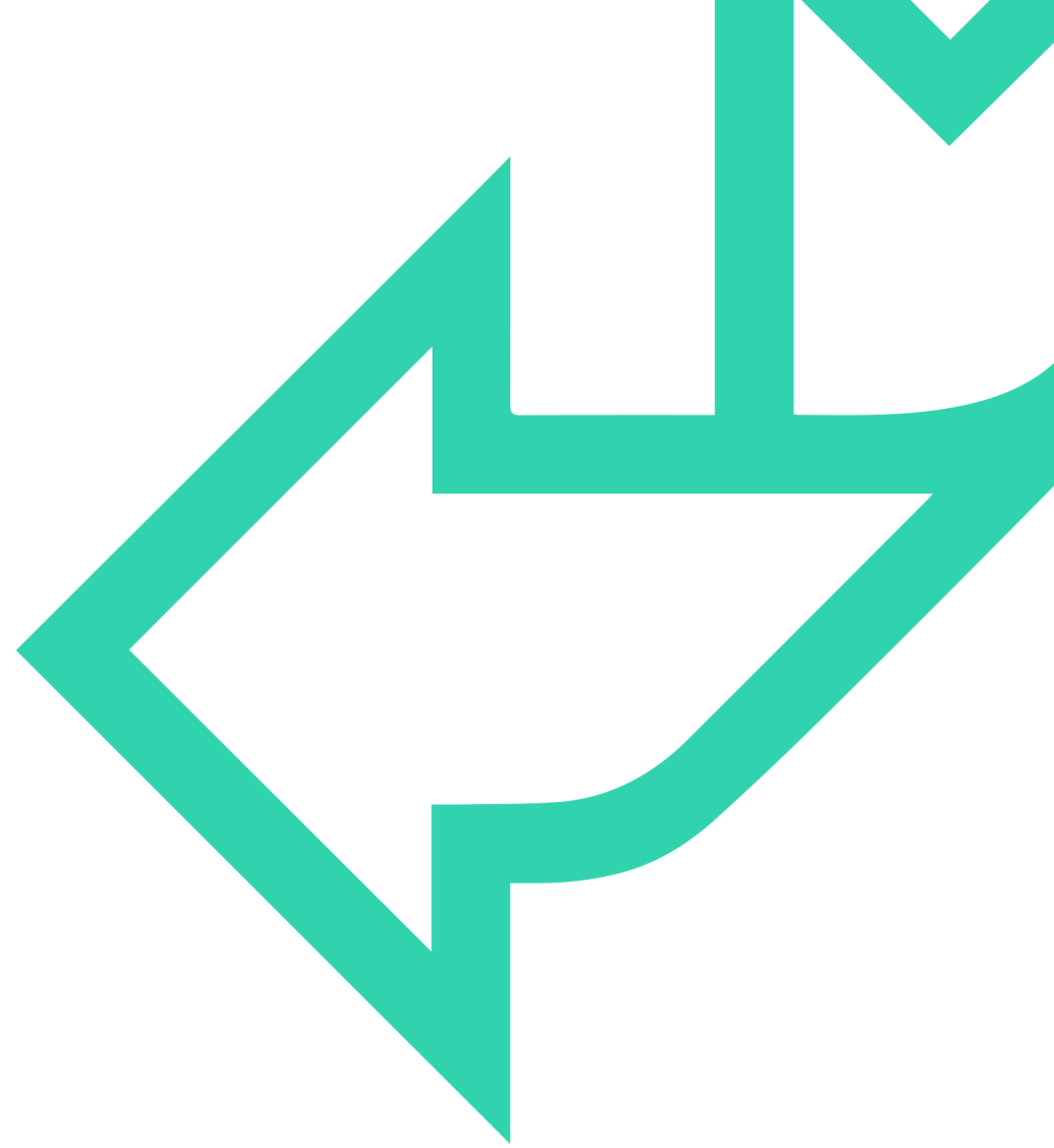# Unit Testing

Module Objectives

- Investigate unit testing

WHAT ARE THE DIFFERENT TYPES OF TEST?

Earlier in this course, we saw there are many types of tests which can be categorised as:

- Unit testing

- Integration testing

- System testing

- Acceptance testing

- Regression testing

Tests should examine non-functional as well as the functional requirements.

In this chapter, we will investigate unit testing, which is essential for testing any IT system.

UNIT TESTING RULES!

Test one thing at a time
- One method.
- One aspect of that method or class.
- Avoid multiple tests inside one unit test.

Avoid if-statements
- If you feel the need for an if-statement, then create more tests!

No dependency between methods in a unit test
- It should not matter in which order the tests are run.

Must not contain hard-coded values unless necessary

Unit tests should be stateless
- Unit tests should not change global data or rely on global data.
- Create all objects needed as new objects.

UNIT TESTING RULES...

Must give the same result every time for a given input
- Relying on databases, files, web-services might change the data source.

Must run fast!
- Important for ci/cd.
- Encourage developers to run these frequently.

Opening files, databases, web services not advised (prohibited)
- Unit tests are not integration tests.
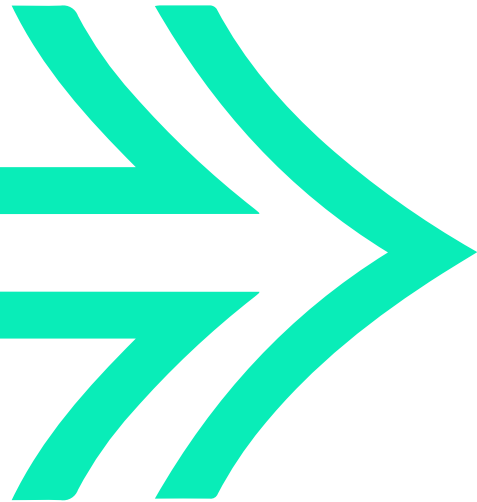- Use a mocking framework (*seen later in this course*).

Test scripts should be as simple as possible
- To read and modify.
- Code units have enough bugs already, don't introduce new ones!
- Write a separate test for each branch of an if statement (two different condition, two tests required).

Unit tests must be...
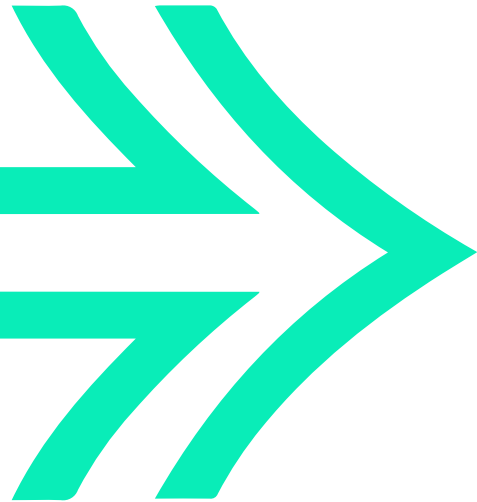
Automatic

- It checks its own results.

Repeatable

- It can be run again with the same results.

Available

- It accompanies the code being tested.

Test structure

Arrange

- Set the starting conditions.

Act

- Invoke the method (or property) that is being tested.

Assert

- Decide if the test has passed or failed.

Assertion-Based Unit Testing Frameworks

'Family' of testing frameworks.

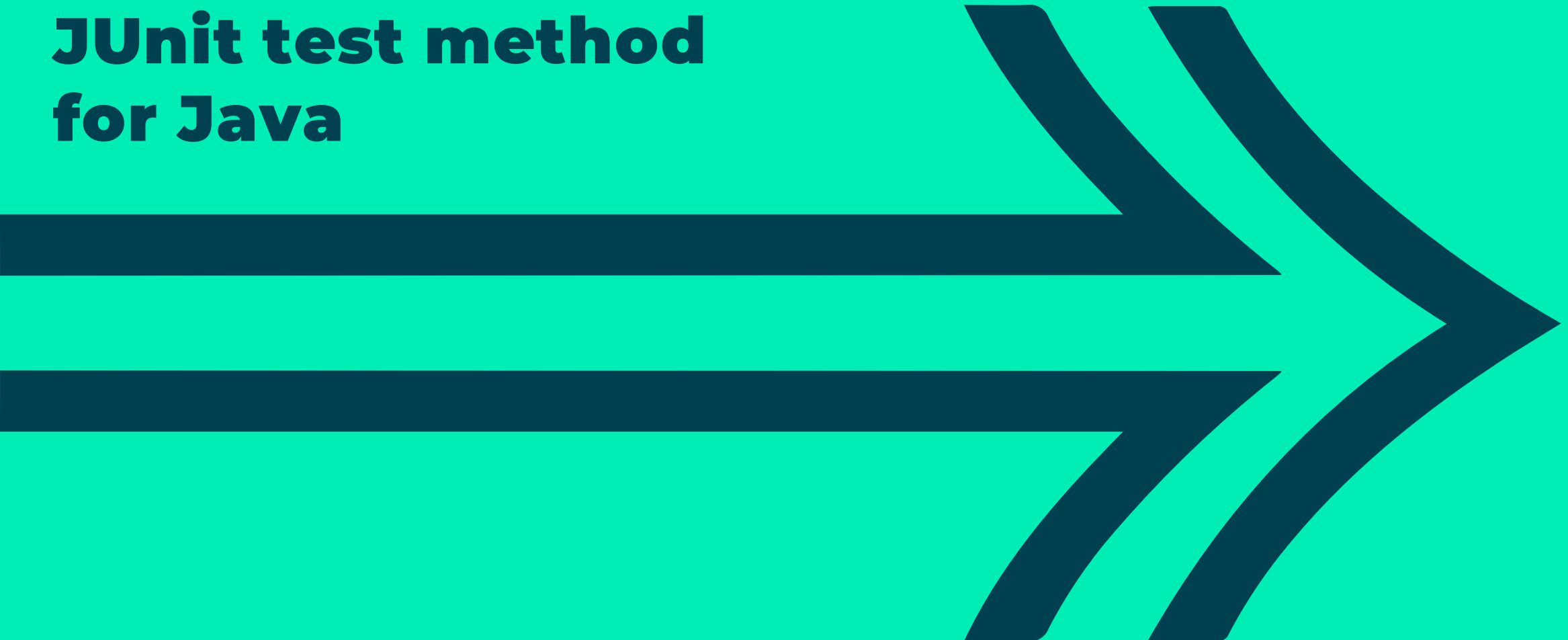- JUnit for Java, NUnit and MSTest for .NET, Test::Unit for Perl

Simple framework with common design to organise and run tests.

- Setup, Test, Assertion, Tear Down

Essential for support of Extreme Programming & Test Driven Development.
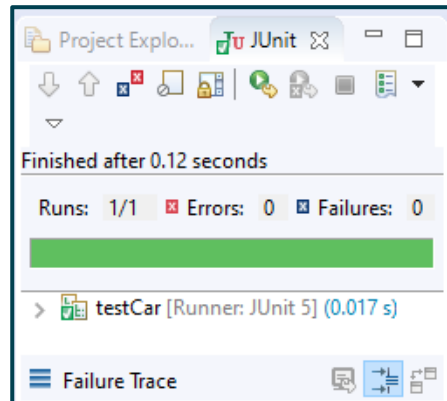
JUnit test method for Java

⟁ How to create a test using Eclipse

1. **Right click on the package name, or class to test, and select**
New > Other > JUnit > JUnit test case

2. **Select "New JUnit Jupiter test" (JUnit 5) at the top of the test wizard and any necessary method stubs. Click Finish once you are happy.**

3. **Run the code.**

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

class TestCar {

        @Test
        void testCarAccelerate() {
                Car car = new Car("Ford");
                car.accelerate(10);
                assertEquals(50, car.getSpeed());
        }

}
```

Project Explo...   JUnit ⊠

Finished after 0.12 seconds

Runs: 1/1   Errors: 0   Failures: 0

> testCar [Runner: JUnit 5] (0.017 s)

Failure Trace

Maven dependency for JUnit 5

- **Projects which use the Maven build tool can include the dependency for JUnit 5 (Jupiter) in the POM file of the project.**

- **Must be nested in <dependencies></dependencies>**

- **Requires the Java version to be set to a minimum of 8, use the maven compiler properties to set this.**

```
<!--
https://mvnrepository.com/artifact/org.junit.jupiter/junit-
jupiter-api -->
<dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-api</artifactId>
    <version>5.9.1</version>
    <scope>test</scope>
</dependency>
```

```
<properties>
    <project.build.sourceEncoding>UTF-
8</project.build.sourceEncoding>
    <maven.compiler.release>11</maven.compiler.release>
    <maven.compiler.target>11</maven.compiler.target>
    <maven.compiler.source>11</maven.compiler.source>
</properties>
```

⚗ JUnit @Before and @After annotations

Marks method to run *before* each `@Test`

Marks method to run *after* each `@Test`

```java
class TestCar {

    Car car;

    @BeforeEach
    public void setUp() {
        car = new Car("Ford");
    }

    @AfterEach
    public void tearDown() {
        car = null;
    }

    @Test
    void testCarAccelerate() {
        System.out.println("@test");
        car.accelerate(10);
        assertEquals(50, car.getSpeed());
    }
}
```

Statuses of a test

- **Passing: ultimately all our tests must pass.**

- **Failing: in TDD we always start with a test which fails.**

- **Erroring: test neither passes nor fails.**
  - Something has gone wrong, a run time error has occurred.

- **Ignored: Using `@Test @Ignore` annotation**
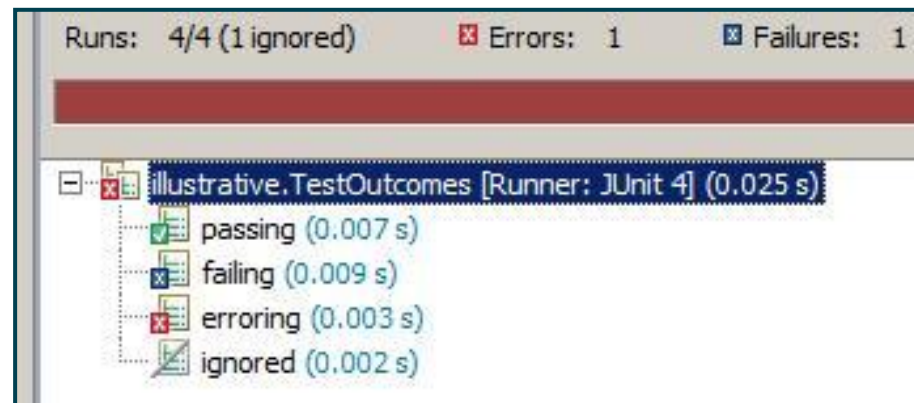
JUnit assertion method 1

Methods are overloaded, e.g.

```
assertEquals(Object expected, Object actual)
assertEquals(long expected, long actual)
assertEquals(String message, Object expected, Object actual)
assertEquals(String message, long expected, long actual)
```

- Use String version: on failure message is displayed.
- Remember order: expected then actual – used in error reporting.

Comparing doubles

```
assertEquals(double expected, double actual)
assertEquals(double expected, double actual, double delta)
```

JUnit assertion method 2

```
assertSame()            – identity of reference
assertNotSame()

assertTrue()            – check Boolean value
assertFalse()

assertNull()            – check if an object is null
assertNotNull()
```

Fail method

```
fail()
fail(String message)
```

Testing Expected Exceptions with JUnit

- **Three approaches to testing for expected exceptions:**
  1. Use the static `Assertions.assertThrows()` method.
  2. Use a `try-catch` block.

(1)

```java
@Test
public void testConstrction() {
    IllegalArgumentException iae =
Assertions.assertThrows(IllegalArgumentException.class, () -> {
        // code that could throw an exception
        User user = null;
        userService.register(user);
    }, "IllegalArgumentException was expected");

    Assertions.assertEquals("Cannot register null object",
iae.getMessage());
}
```
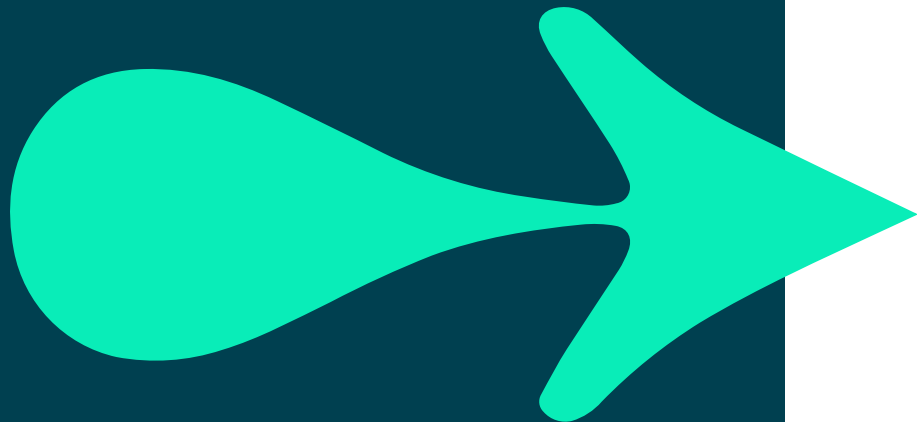
(2)

```java
@Test
public void testExpectedException3() {
    try {
        new Employee("Fred", -1);
        fail("Should raise exception");
    } catch (IllegalArgumentException e) {
        assertThat(e.getMessage(), containsString("Invalid age"));
    }
}
```

EXERCISE

- Please see your Exercise Guide and complete Exercises 1 and 2.

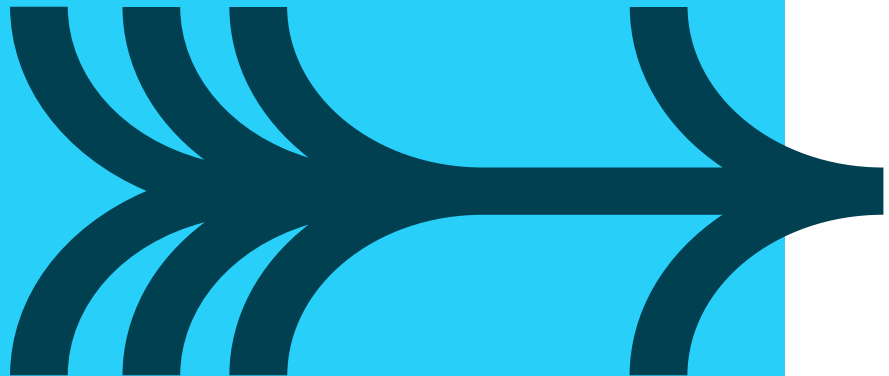- Develop and write unit tests in Java, C#, Python and JavaScript..

TDD

Test-driven Development

TEST-DRIVEN DEVELOPMENT
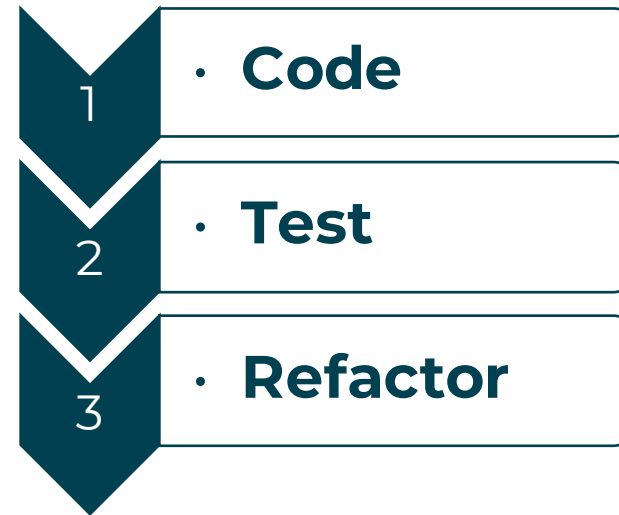
It is an evolutionary approach…

- You write test before you write code.
- Run test to detect defect, then refactor.
- Repeat the process until sufficiently sure of correctness.
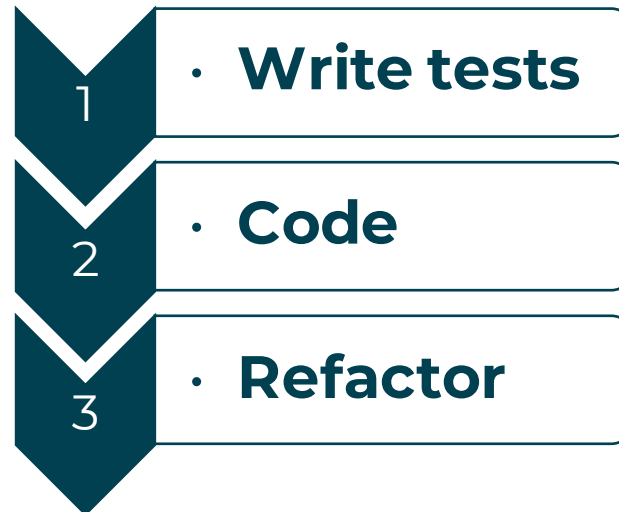
What is the goal of TDD?

- One view says it is specification and design – not validation.
- It is a way of thinking through design before coding to functionality.
- Another view says it is a programming technique.
- The goal is to write clean and robust code that works.
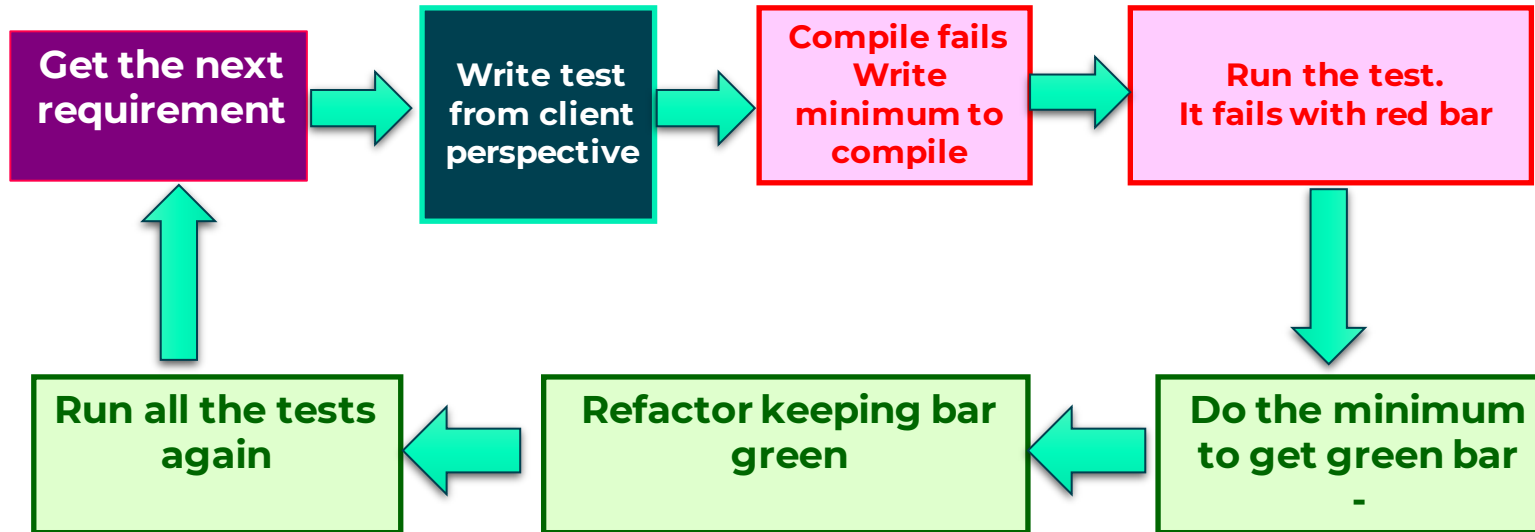- Both arguments have merit.

TDD

Non-TDD software development:

1. · **Code**
2. · **Test**
3. · **Refactor**

The TDD way:

1. · **Write tests**
2. · **Code**
3. · **Refactor**

TDD Cycle

Get the next requirement → Write test from client perspective → Compile fails Write minimum to compile → Run the test. It fails with red bar → Do the minimum to get green bar - → Refactor keeping bar green → Run all the tests again → (back to Get the next requirement)
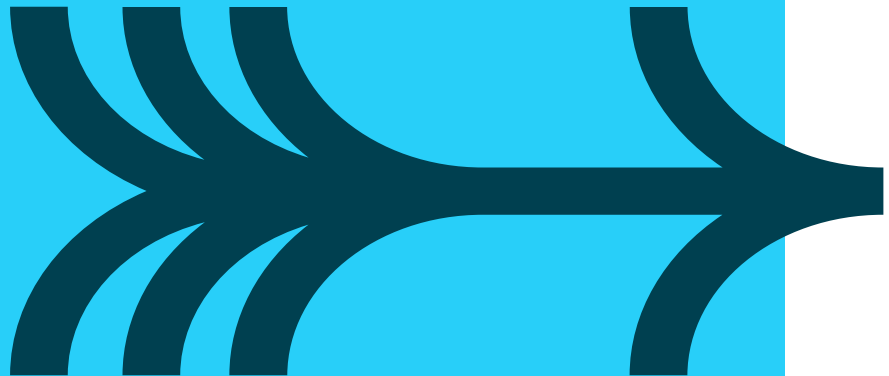
WHY USE TDD?

- Helps developers **understand** and cover all the **requirements**.

- Is an **iterative** development (develops in small chunks).

- **Catches defects early**.

- **Forces** developers to **write test cases!**

- Helps with the **design** of code.

- Acts as **documentation** for code.

POINTS TO REMEMBER

- **Think about what you are trying to accomplish,** not just writing an individual test.

- **Think of an individual test in terms of expected behaviour** instead of just verifying some inputs and outputs.
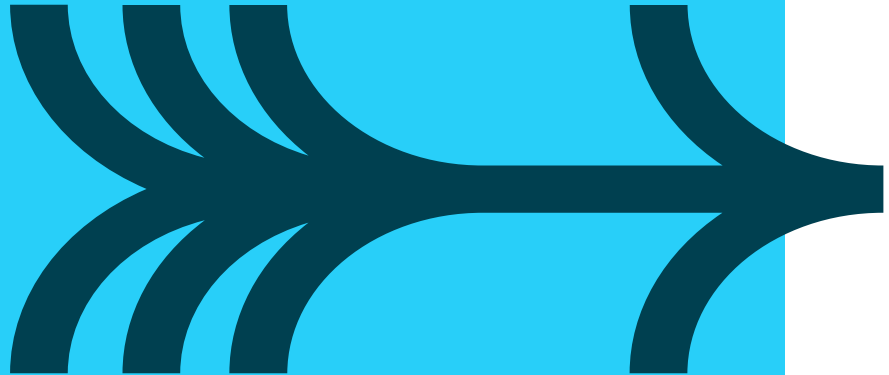
Ask yourself:

- How would I know the call has worked?

- What would I expect it to do?

- This determines what kind of assertions you need.
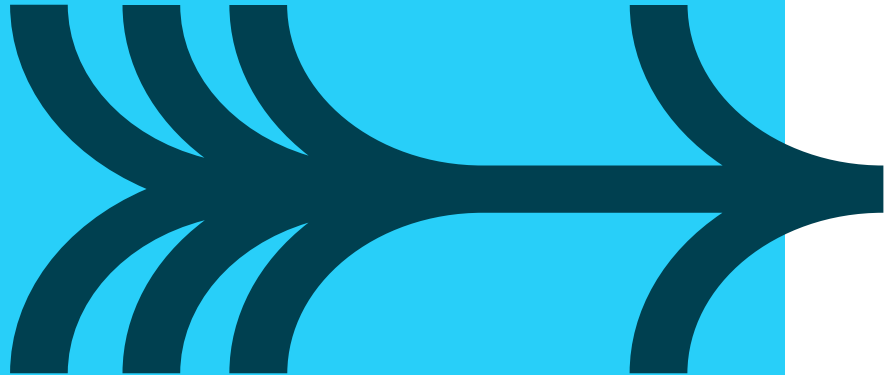
POINTS TO REMEMBER

External dependencies you need
- Use abstractions instead (dependency inversion principle).
- We will investigate this topic later.

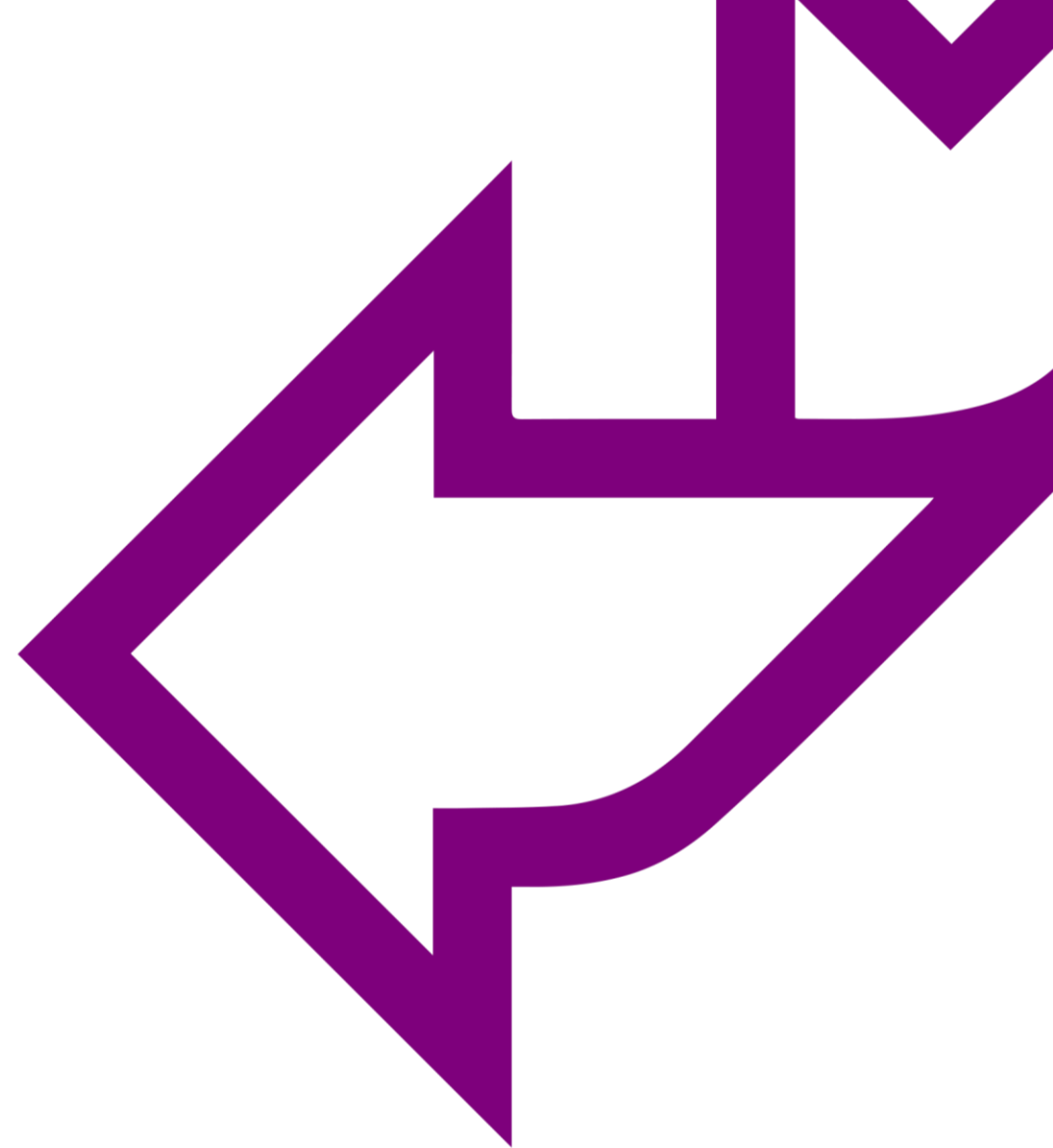**Check for the functions side effect**

Always:
- Write your test.
- Watch it fail.
- Code to pass.
- Refactor.

This is non-negotiable!

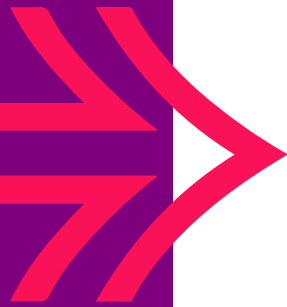# Mocks and Stubs

- **Using MOQ**

OBJECTIVES

In this chapter, you'll learn about creating and using **mocks** and **stubs**.

STUBS, MOCKS AND FAKE OBJECTS

- Used to create a class that returns data to a test.
- Data should be easily created and will always stay the same.
- They don't show how a class interacts with the system.
- Just provides data.
- MOQ frameworks can create Stubs.
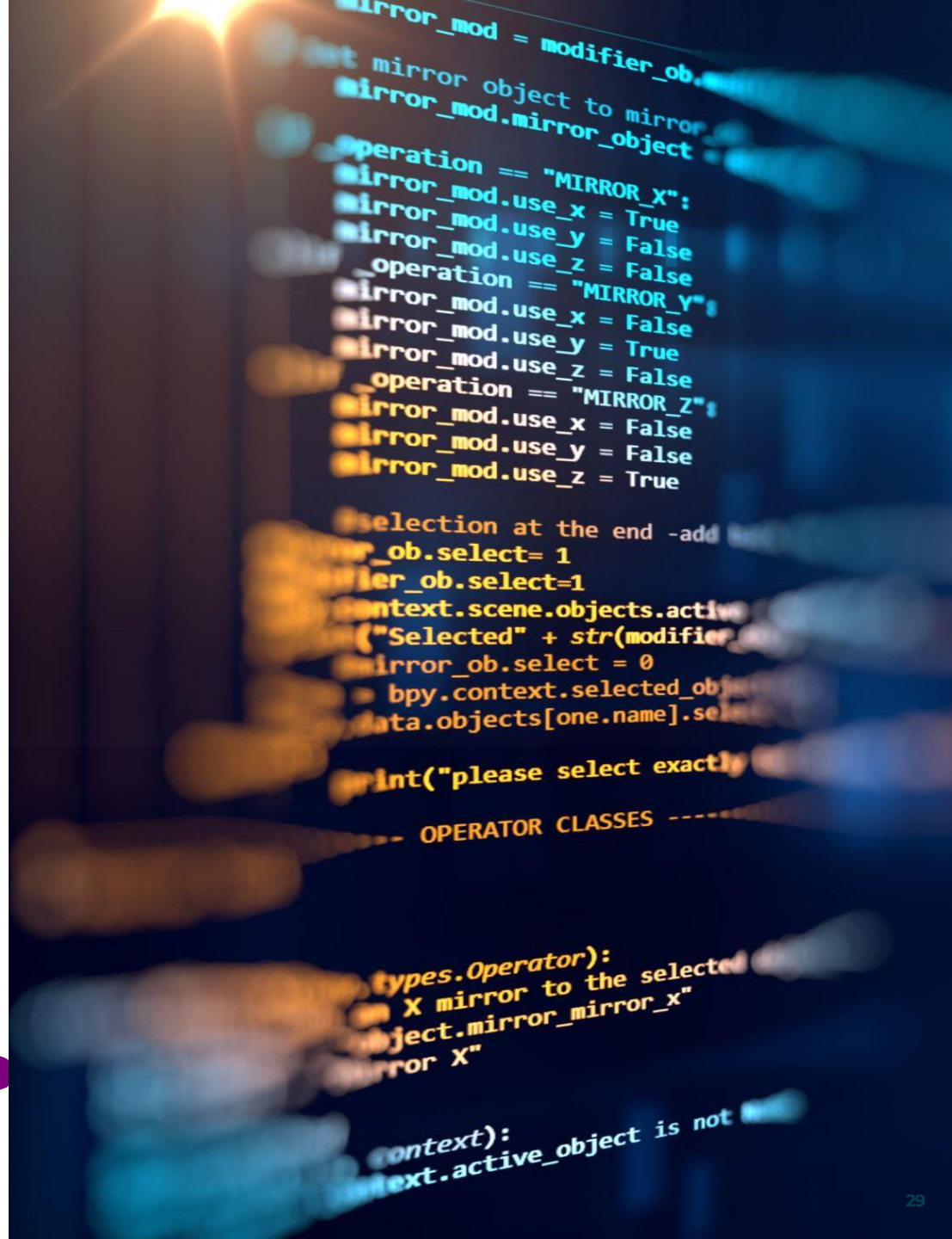
Stubs, mocks and fake objects

Mock:

- When your class interacts with the system.

- Created using MOQ frameworks.

- Can provide the same functionality as the Stub, but Stubs are easier to create.

- They could create the same functionality as the class under test.

Fake:

- Stands for the real object.

- Use this when creating the mock object is too hard. For example, when it supports events.

Creating mocks and stubs with Java

Using Mockito with Java

- Mockito is a very useful app for Java developers to mock external dependencies.
- To use **Mockito**, you need to set it up for you application.
- See here for a Maven project's POM file.

```
<dependencies>
  <dependency>
      <groupId>org.junit.jupiter</groupId>
      <artifactId>junit-Jupiter-
api</artifactId>
      <version>5.9.1</version>
      <scope>test</scope>
  </dependency>
  <dependency>
      <groupId>org.mockito</groupId>
      <artifactId>mockito-core</artifactId>
      <version>4.6.1</version>
      <scope>test</scope>
  </dependency>
```

```
  <dependency>
      <groupId>org.mockito</groupId>
      <artifactId>mockito-junit-
jupiter</artifactId>
      <version>4.6.1</version>
      <scope>test</scope>
  </dependency>
<dependencies>
```

Using Mockito - setting expectations

- Create an interface for the actual system that your application depends on.

- Then ask Mockito to create a mock and setup the methods' expected values.

- You can create a Stub, but you will have to code it and maintain it in your project.

```java
public class StubDatabase implements QADatabase {
  public String getUsernameByID(int id) {
    String[] names = {"Bob","Anna","Mike","David","Lily", "Fred", "Kimberly"};

    if (id < names.length)
        return names[id];
    else
        return null;
    }
}
```

So, how do we get Mockito to do all this for us? Let's see...

Mocking an object using Mockito

```java
public interface QADatabase {
    public String getUsernameByID(int id);
}
```

Object to mock

Class requiring the mock

```java
public class QAController {
    private QADatabase qaDb;

    public QAController(QADatabase db) {
        this.qaDb = db;
    }
}
```

```java
@ExtendWith(MockitoExtension.class)
public class FirstTest {

    @Mock
    QADatabase db;

    @InjectMocks
    QAController controller;

    @Before
    public void setUp() {
        Mockito.when(db.getUsernameByID(0)).thenReturn("Bob");
    }
}
```

Exercise

- Please see your unit testing exercise guide.