QA
POWERING
POTENTIAL

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 or
- 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

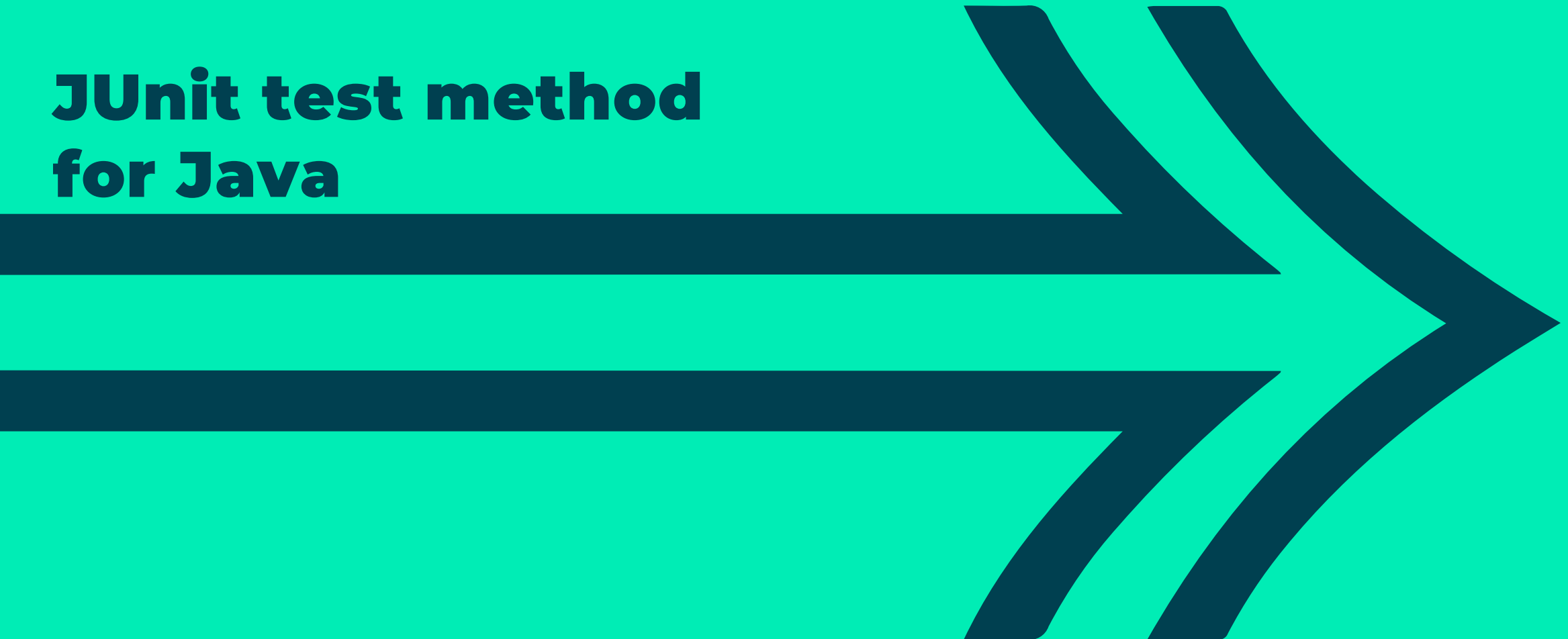
- JUnit for Java, NUnit and MSTest for .NET, Test::Unit for Perl

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



JUnit @Before and @After annotations

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

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

```
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

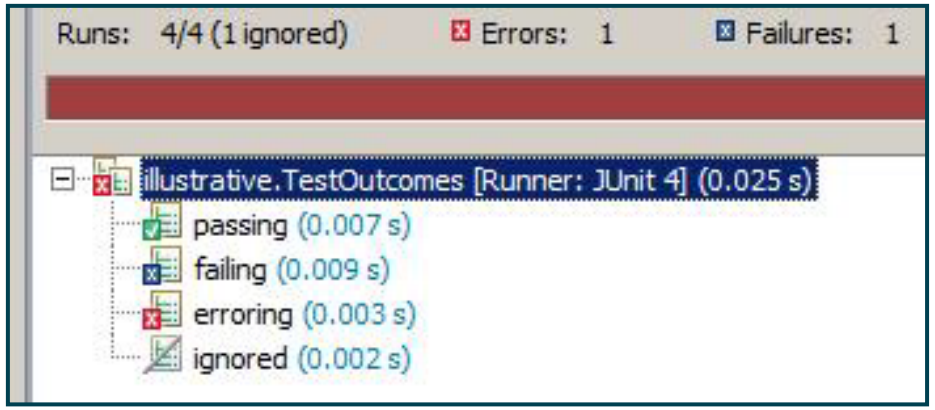
Passing: ultimately all our tests must pass

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

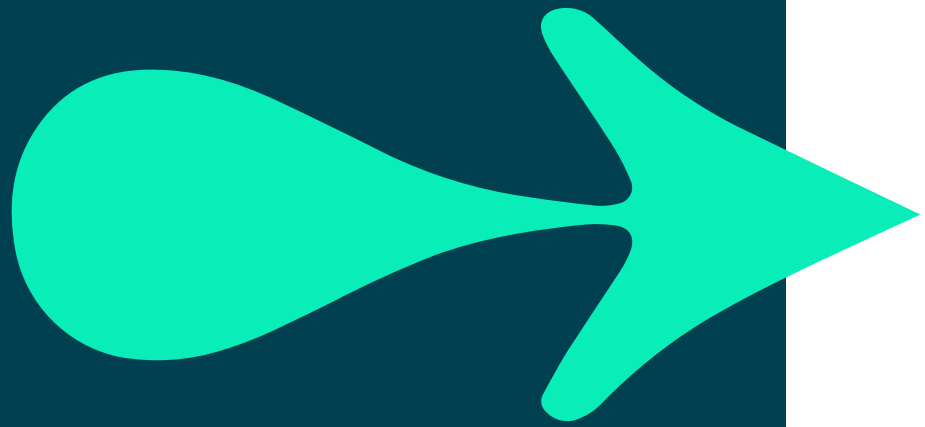
1

```
@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

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



EXERCISE

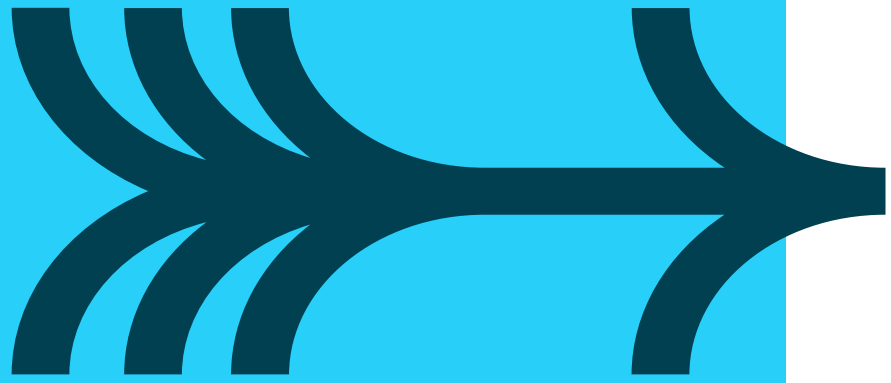
- Please see your Exercise Guide and complete exercises 1 and 2
- Develop and write unit tests in Java, C#, Python and JavaScript.

POWERING POTENTIAL

TDD

Test-driven Development

TEST-DRIVEN DEVELOPMENT



It is an evolutionary approach...

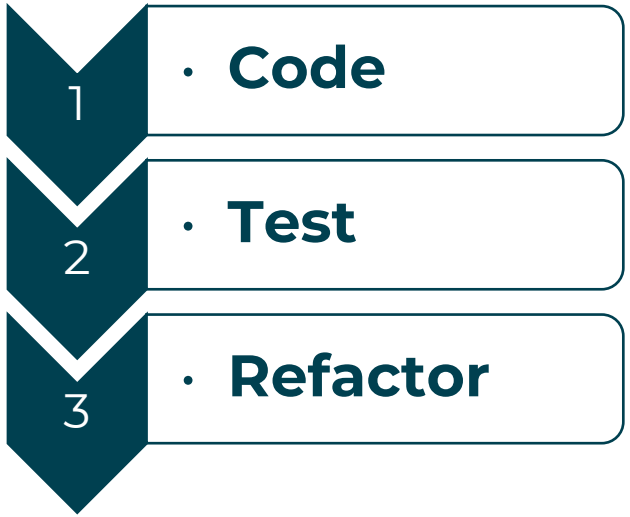
- You write test before you write code
- Run test to detect defect, then refactor
- Repeat the process until sufficiently sure of correctness

What is the goal of TDD?

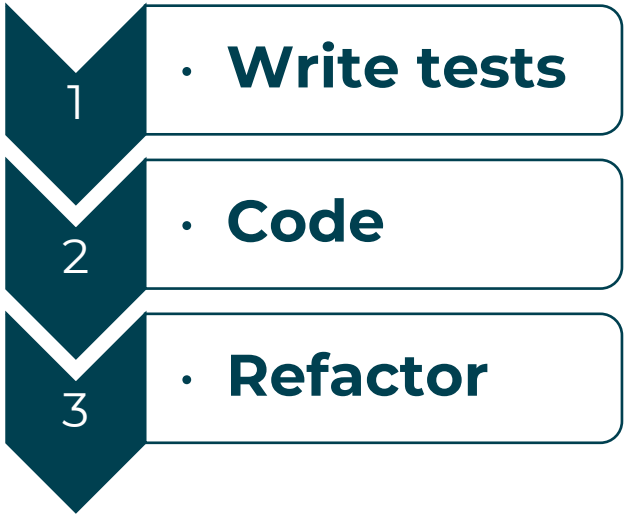
- One view says it is specification and design – not validation
- It is a way of thinking through design before coding to functionality
- Another view says it is a programming technique
- The goal is to write clean and robust code that works
- Both arguments have merit

TDD

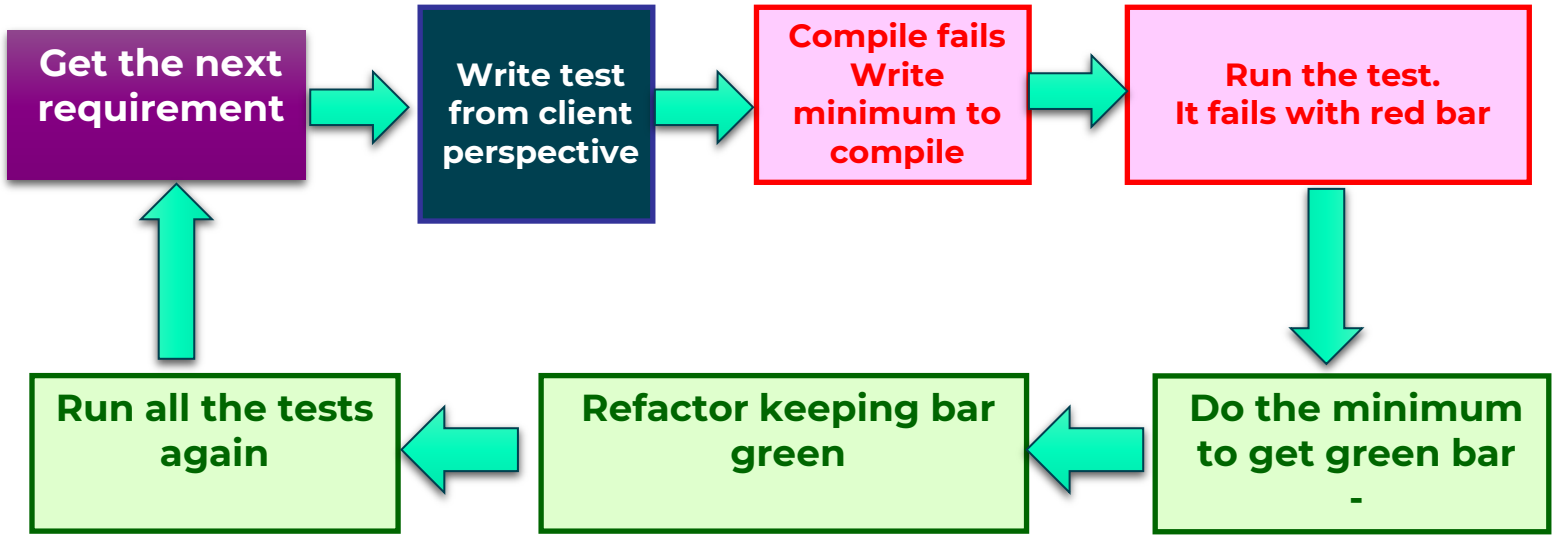
Non-TDD software development:



The TDD way:

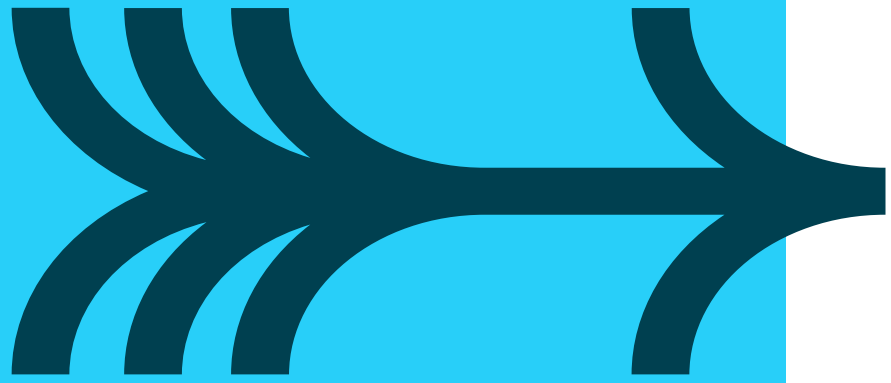


TDD Cycle


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

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



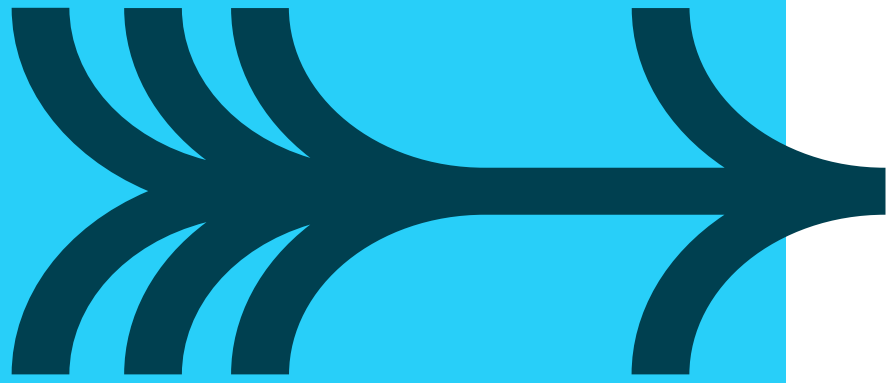
POINTS TO REMEMBER

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

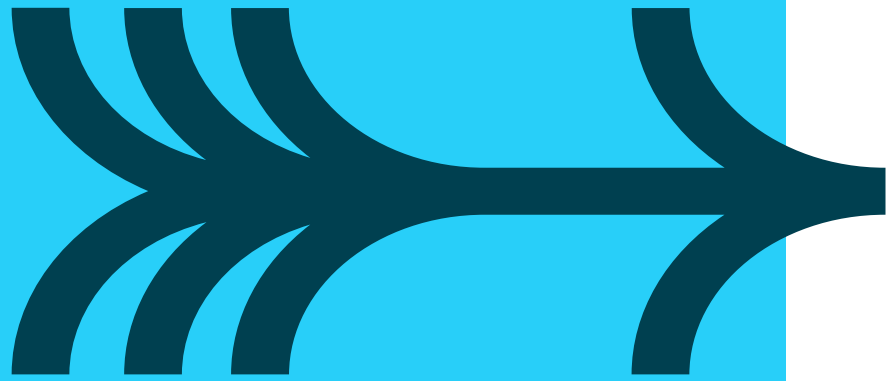
- **Think of an individual test in terms expected behaviour** instead of just verifying some inputs and outputs

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!