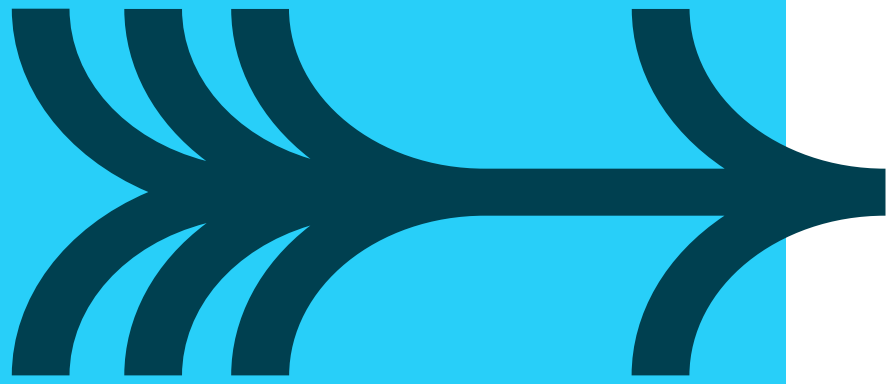
POWERING POTENTIAL

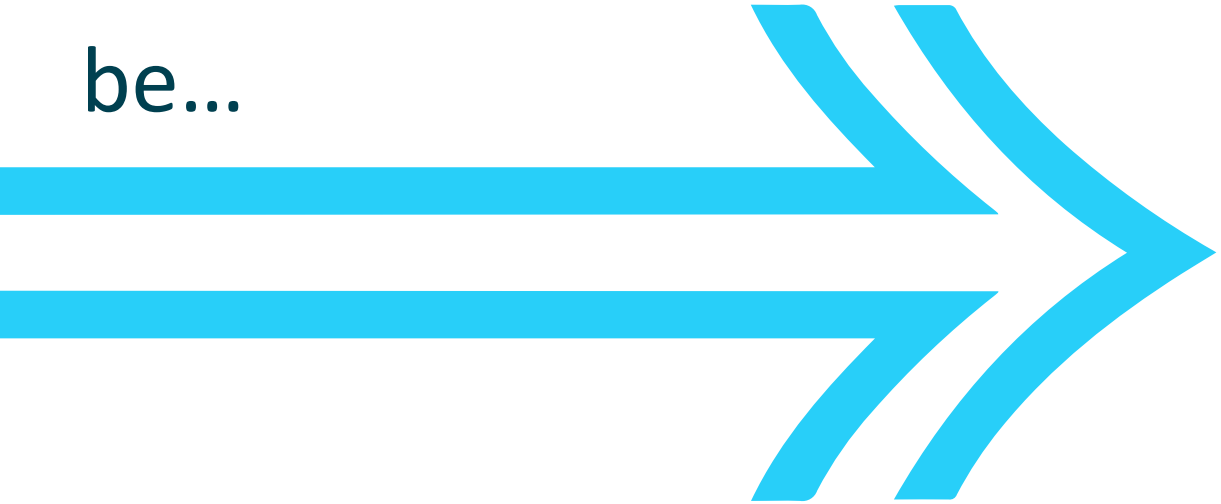
Introduction to Testing

CONTENTS



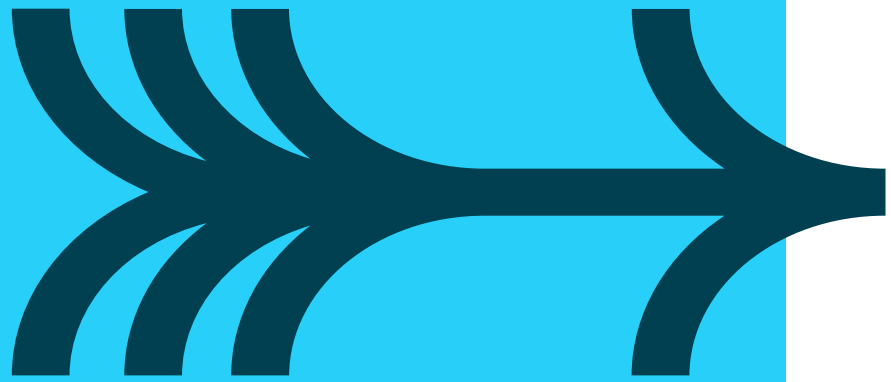
- **Objectives**
  - Look at the main testing framework used in Java development
- **Contents**
  - JUnit
    - How to set up and run
    - Annotations
- **Hands on lab**
  - Author Unit Tests

Unit Tests must be...



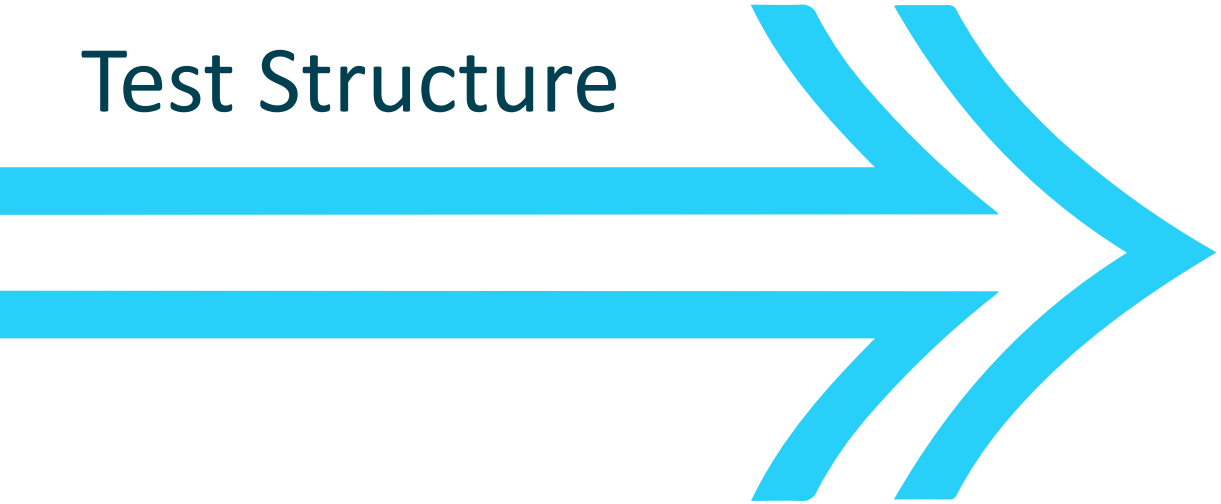
- **Automatic**
  - it checks its own results
- **Repeatable**
  - it can be run again with the same results
- **Available**
  - it accompanies the code being tested

THE BENEFITS ARE REALLY FOR THE DEVELOPER



- You fix all the trivial problems as you go along
- You know that they have not recurred
- You document without effort, how you see other s/w interfacing with yours
- You are able to refactor your code to make it more maintainable, faster… knowing that you haven’t broken anything.

Test Structure

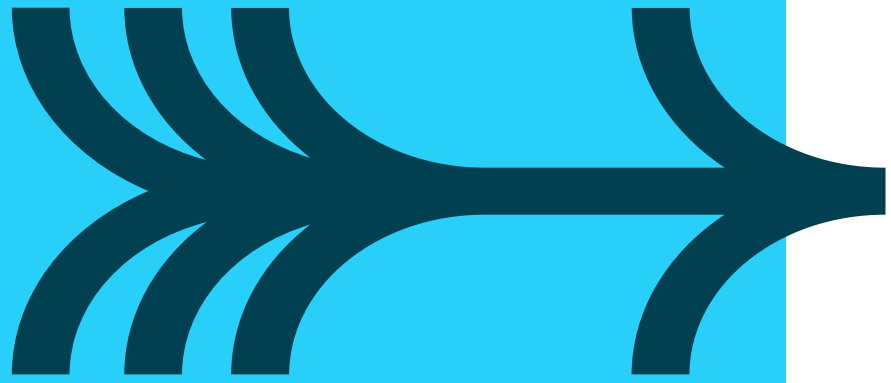


- **Arrange**
  - Set the starting conditions
- **Act**
  - Invoke the method (or property) that is being tested
- **Assert**
  - Decide if the test has passed or failed

Manual Tests

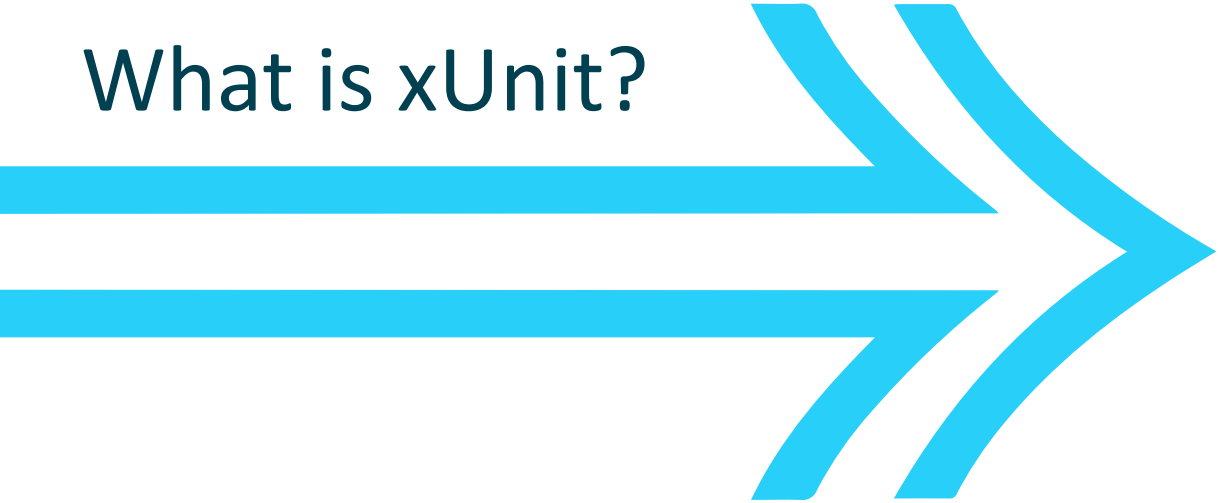
- **Write a test harness for the Class Under Test (CUT)**
  - Main method creates instance of class, invokes methods and outputs to the results to the console
- **Drawbacks**
  - Not structured; have to hand-craft each time
  - Not necessarily repeatable; may not work in 2 weeks time
    - Should be able to run at click of button and see whether they passed or failed
  - Will not run all the code
  - No standardised reporting
    - Requires visual inspection of console output
    - You may miss failures
  - Integration with other tools (e.g. your build, code coverage)

UNIT TESTING



- **Unit tests**
  - Test one unit in isolation
  - Also known as Component or Module testing
- **What is a unit?**
  - Method
  - Class
  - Database query or transaction
  - Web Page
- **What are you testing?**
  - You know the internals of the test – “White Box”

What is xUnit?



- **“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?

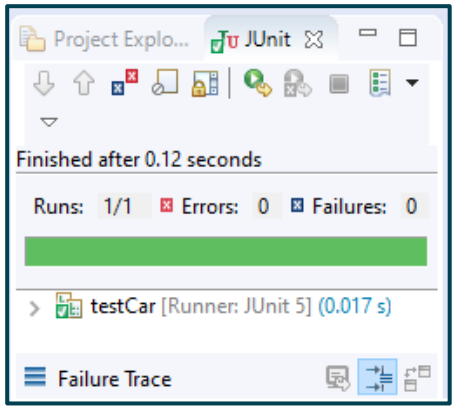
- **Right click on the package name and select**
  - New > Other > JUnit > JUnit test case

- **Select the CUT in the dialog**

  and then write code:

```
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());
        }
}
```

- **Run the code**



JUnit @Before and @After annotations

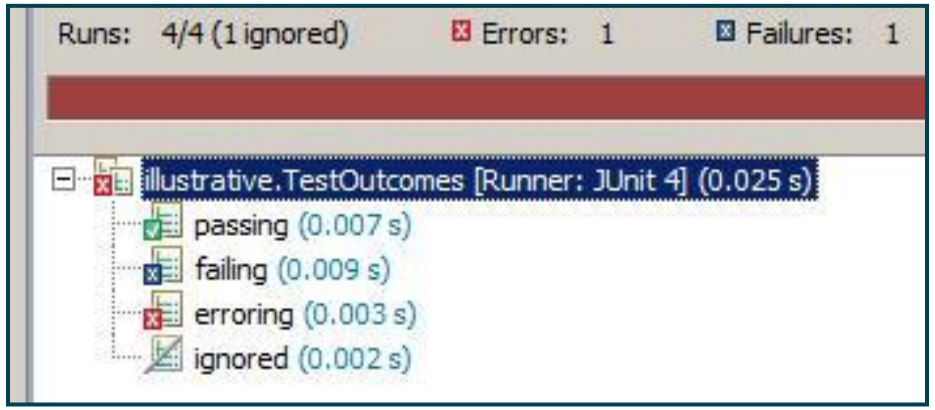
Marks method to run *before* each `@Test`

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 Assertions methods 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 methods 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)
```

JUNIT @TEST ANNOTATION

- `@Test` marks method as a unit test
- `@Test(expected = Exception.class)`
  - Will fail if the method does not throw the expected exception

```
@Test(expected =
IndexOutOfBoundsException.class)
```

- `@Test(timeout = 200)`
  - Will fail if the method takes longer than 200 milliseconds

Testing Expected Exceptions with JUnit

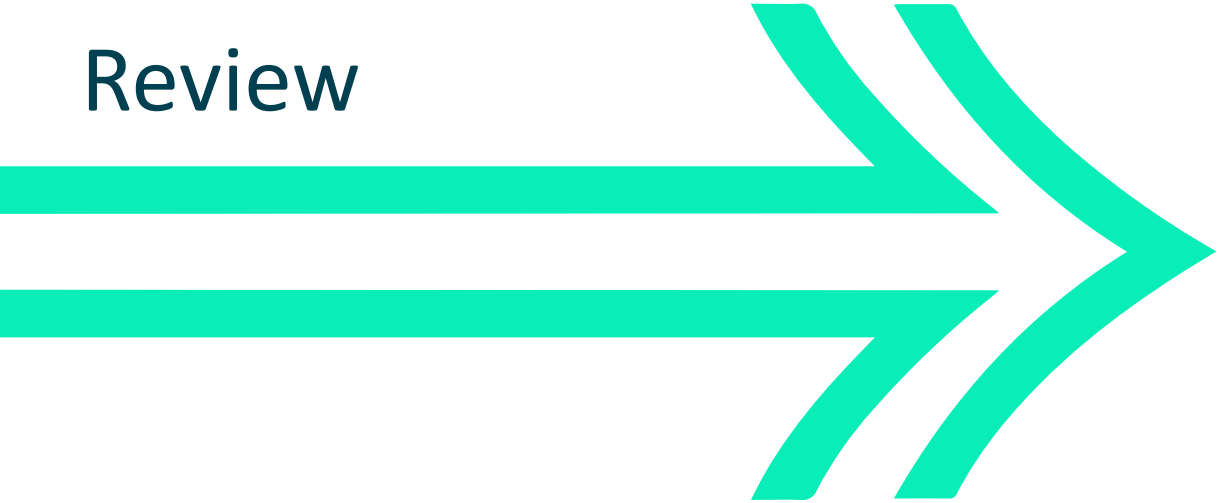
- **3 approaches to testing for expected exceptions**
  - Use the **`ExpectedException`** rule
  - Use the **`expected`** parameter with **`@Test`**
  - Use the **`try-catch`** block

```
@Rule
public ExpectedException exception = ExpectedException.none();

@Test
public void testConstrction() {
   exception.expect(IllegalArgumentException.class);
   exception.expectMessage(containsString("Invalid age"));
   new Employee("Fred", -1);
}
```

```
@Test(expected = IllegalArgumentException.class)
public void testConstrction() {
   new Employee("Fred", -1);
}
```

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

Review



- Unit Testing and Test Driven Development are the recommended approach to produce quality software
- JUnit encourages the TDD mindset

Hands On Lab

- Writing tests for a security checker class for userID / password validation