POWERING POTENTIAL

TESTING

UNIT TESTING

What are Unit Tests?

A software testing method that tests individual sections of source code

- **Unit** = the smallest testable part of the software
- Allows us to **validate units** in isolation
- **Test-driven development** (TDD) – written before the code to be tested
- **Automated**
- **Developers** write, execute and maintain test code

BENEFITS OF UNIT TESTING

- **Shows you that the code works as you develop it**
- Detect **bugs** or defects early on
  - Easily **localise** the source of the bug
- **Continuous integration** (CI) makes deployment easier
- **Confidence** in code quality and functionality

KEY CONCEPTS

- Core practice of **XP**
- Can be adopted within other methodologies
- **TDD:** test written before implementation and tests drive API design
- **Automated and self-validating**
- **Easy to maintain**

F.I.R.S.T.

Unit tests must be...

- **F**ast
- **I**ndependent
- **R**epeatable
- **S**elf-validating
- **T**imely

- Robert Martin, *Clean* Code, 2009

RIGHT B.I.C.E.P

Right: Are the results right?

B: Are all the **b**oundary conditions correct?

I: Can you check the **i**nverse relationships?

C: Can you **c**rosscheck results using other means?

E: Can you force **e**rror conditions to happen?

P: Are **p**erformance characteristics within bounds?

READABLE TESTS: CODING BY INTENTION

Four phases:

1. **Setup / Arrange**: set up the initial state for the test.
2. **Exercise / Act:** perform the action under test.
3. **Verify / Assert:** determine and verify the outcome.
4. **Clean-up:** clean up the state created.

Each phase should be:

- Clearly expressed, including your expected outcomes
- Well documented

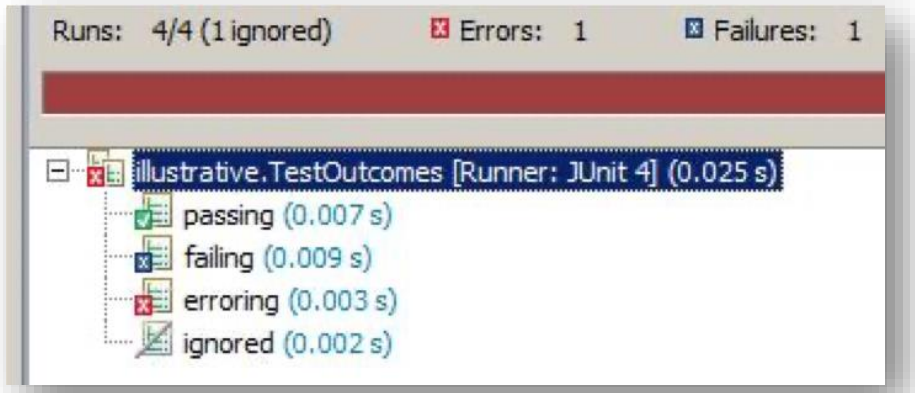
TEST STATUSES

Passing: ultimately, all our tests must pass

Failing: in TDD, always start with a test which fails

Erroring: test neither passes nor fails
Something has gone wrong like a run-time error

Ignored: `@Test` `@Ignore`



Manual Testing vs Automated Testing

Manual Testing	Automated Testing
Only certain people can execute the tests	Anyone can execute the test
Difficult to consistently repeat tests	Perfect for regression testing
Manual inspections can be error prone and aren't scalable	Series of contiguous testing, where the results of one test rely on the other
Doesn't aggregate, indicate how much code was exercised, or integrate with other tools (e.g. build processes)	The build test cycle is increased

Unit vs Component vs Integration

Unit Testing	Component Testing	Integration Testing
Ensures all of the features within the Unit (class) are correct	Similar to unit testing but with a higher level of integration between units	Involves the testing of two or more integrated components
Dependent / interfacing units are typically replaced by stubs, simulators or trusted components	Units within a component are tested as together real objects	
Often uses tools that allow component mocking / simulation	Dependent components can be mocked	

REVIEW

Please view **04-A Unit testing code.pptx**

if you need to review unit testing in Java and C#

LABS

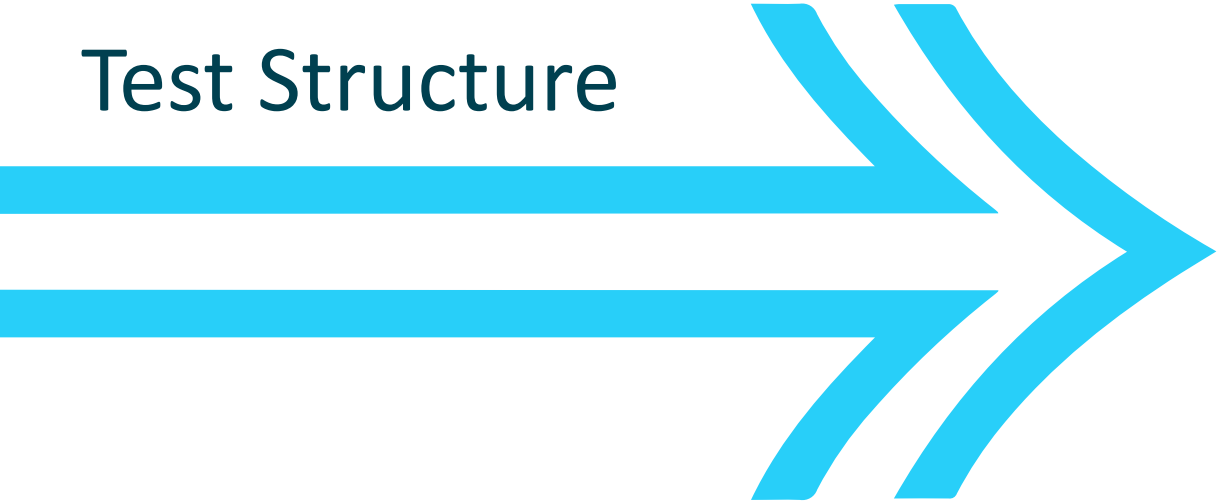
"Unit Testing" lab

+

"Readable Tests" lab

Unit testing revision material

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

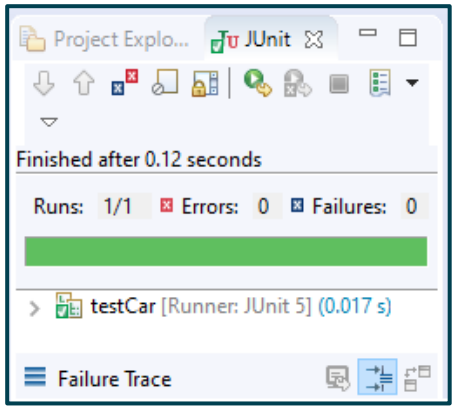
How to create a test?

- **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());
    }
}
```

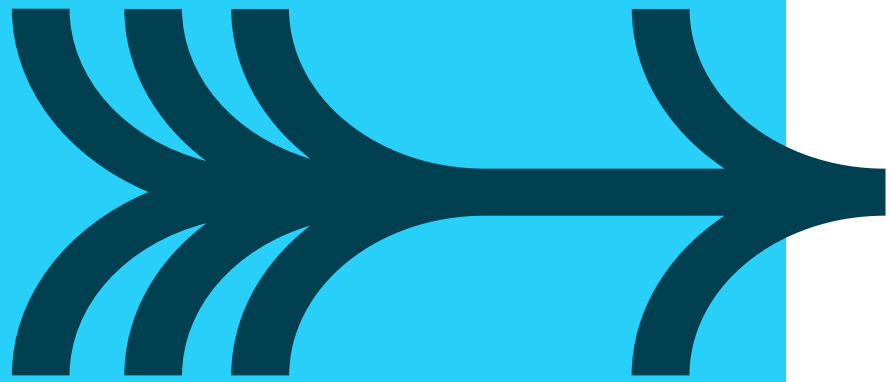
JUnit Assertion methods 2

```
assertSame()          – identity of reference (Are they the same objects?)
assertNotSame()

assertTrue()          – check Boolean value
assertFalse()

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

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

JUNIT

TESTING EXPECTED EXCEPTIONS



- `@Test` marks method as a unit test
- `@Test(expected = Exception.class)`
  - Fails if the expected exception is not thrown

  `@Test(expected = DateTimeException.class)`
- `@Test(timeout = 200)`
  - Fail if the method takes longer than 200 ms

Testing Expected Exceptions with Junit 5

```java
@Test
void testWithdrawWithInsufficientFunds() {

    Account acc = new Account(123, 100, "Bob");

    assertThrows(InsufficientFundsException.class, () -> {
        acc.withdraw(1000);
    });

}
```

```java
public class InsufficientFundsException extends Exception {

        InsufficientFundsException(String message) {
                super(message);
        }
}
```

POWERING POTENTIAL

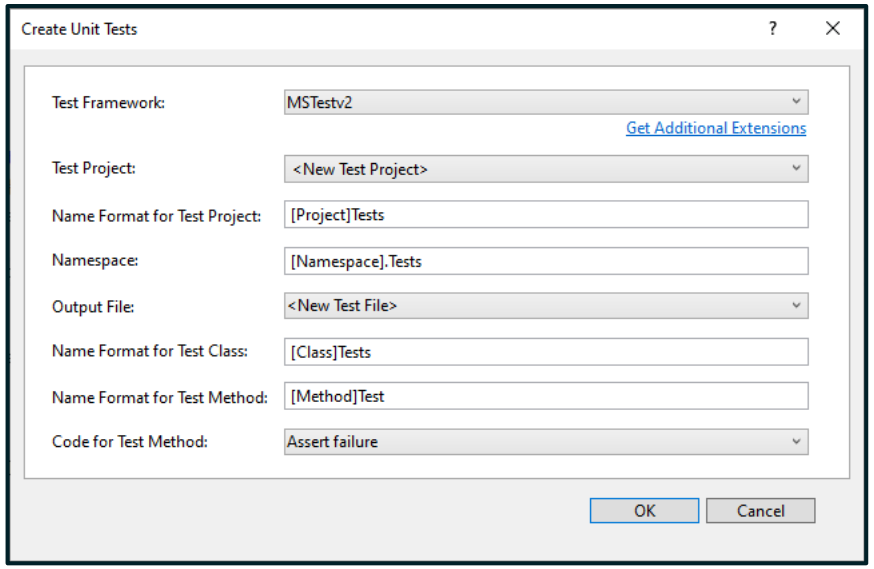
Unit testing method for .NET

Create a MS-Test project

- Right mouse click on a method and then create a unit test project

```
public class Car
{
    public int Speed { get; set; }
    private string Model { get; set;
    public void accelerate(int second
    {
        Speed += (5 * seconds);
    }

    public Car(string model)
    {
        Model = model;
    }
```

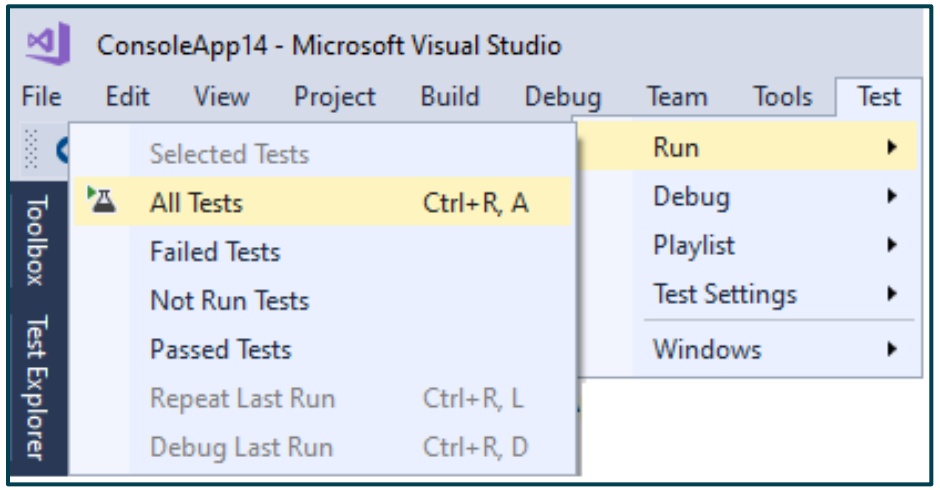
- Quick Actions and Refactorings...
- Rename...
- Remove and Sort Usings
- Peek Definition
- Go To Definition
- Go To Implementation
- Find All References
- View Call Hierarchy
- Create Unit Tests

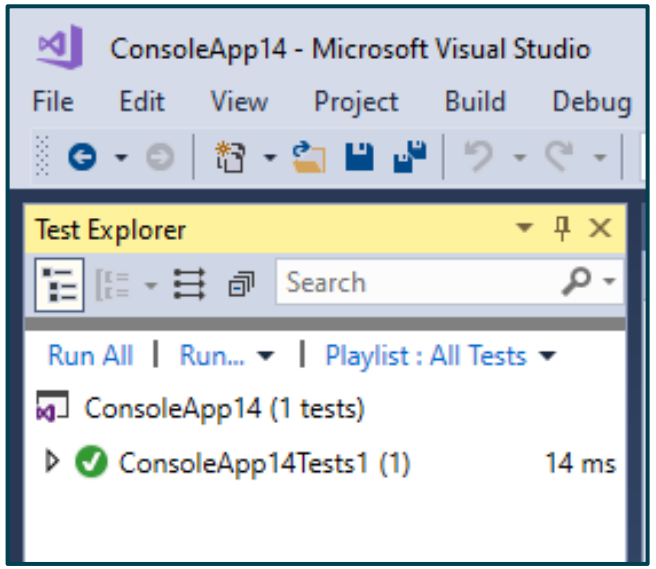


Write test code

```
[TestClass()]
public class CarTests {
   Car car;

   [TestInitialize]
   public void SetUp() {
       car = new Car("Ford");
   }
   [TestCleanup]
   public void TearDown() {
       car = null;
   }
   [TestMethod()]
   public void accelerateTest() {
       car.accelerate(10);

       Assert.AreEqual(50, car.Speed);
   }
}
```

Run the tests





Testing Expected Exceptions with MS-Test

- **The following test succeeds if null values entered as username**
- **The Login() constructor throws exception**
- **We test that exception is thrown**

```
[TestMethod]
[ExpectedException(typeof(ArgumentException))]
public void TestNullUsername()
{
        LogonInfo login = new Login(null, "password123");
}
```

Review

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