Testing code

Testing Patterns

QA

Learn. To Change.

Objectives

During this session you will explore different testing patterns that you can apply to your tests

Assertions

Expressions that encapsulate a testable logic about the product you're testing.

Types of Assertion

Resulting state assertion

Resulting state assertion is a standard test; the state that we expect.

```
@Test
public void testAdd() {
  Calculator calc = new Calculator();
  assertEquals(5, calc.add(2, 3), "2 + 3 should equal 5");
}
```

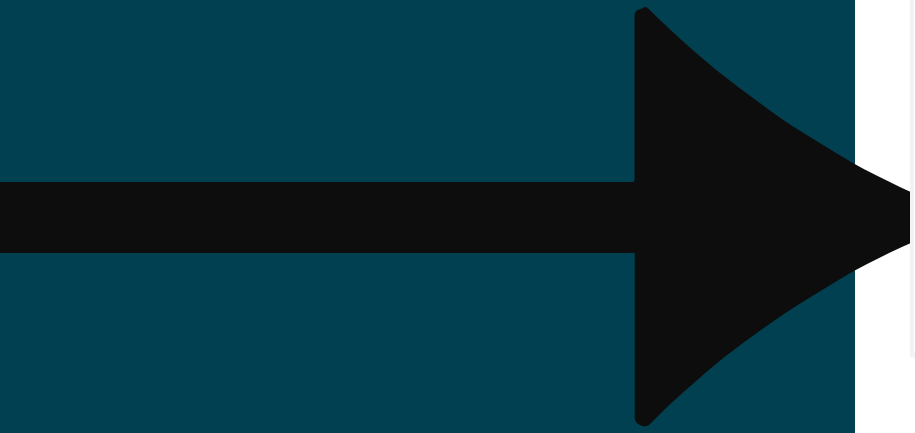
Later, we will explore an alternative style in the BDD section of this course.

Types of Assertion

Guard assertion

Asserts a precondition for the test to be correct,
and follow it with the resulting state.

```
public static void processOrder(int quantity) {
    // Guard condition to ensure quantity is valid
    if (quantity <= 0) {
      throw new IllegalArgumentException("Quantity must be > 0");
    }
    // more code
```



```
@Test
void testInvalidQuantityThrowsException() {
   // Exception should be thrown for invalid quantity
   Exception exception =
       assertThrows(IllegalArgumentException.class,
            () -> OrderProcessor.processOrder(-5));
```

Types of Assertion

Delta assertion

If you can't guarantee the absolute resulting state, test the delta (difference) between the initial and resulting states.

```
@Test
void testDivideWithDelta() {
        double result = Calculator.divide(10, 3);
        double expected = 3.3333;
        double delta = 0.0001; // Allowable tolerance

        assertEquals(expected, result, delta,
                "Result should be approximately 3.3333");
}
```

```
C#
Assert.AreEqual(expected, result, delta, "Message");
```

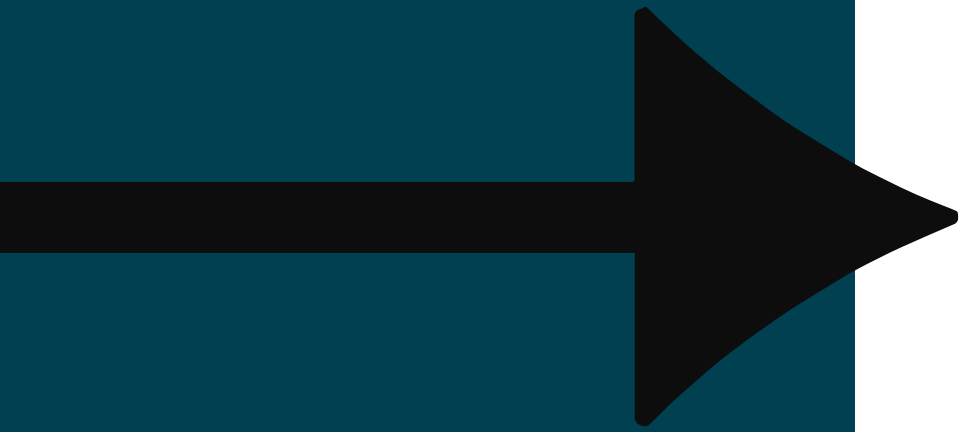
Types of Assertion

Custom assertion

Helps your test code respect DRY
('don't repeat yourself', i.e. no duplications)

```
class QA {
  public static void assertApproxEquals(
                double expected, double actual, double delta,
                String message) {

        assertTrue(Math.abs(expected - actual) < delta,
        message + " Expected: " + expected + " but got: " +
                actual + " within tolerance of: " + delta);
  }
}
```



```
@Test
void testDivide_WithCustomAssertion() {

    double result = Calculator.divide(10, 3);
    QA.assertApproxEquals( 3.3333, result, 0.0001,
                        "Division result mismatch!");
}
```

Parameterised tests

Parameterised unit tests allow you to run the same test multiple times with different input values, improving test coverage and reducing code duplication.

Using Java in Maven: **Create a maven project and then add JUNIT5 to your POM file**

```
<dependencies>
<!-- https://mvnrepository.com/artifact/org.junit.jupiter/junit-jupiter -->
    <dependency>
        <groupId>org.junit.jupiter</groupId>
        <artifactId>junit-jupiter</artifactId>
        <version>5.12.1</version>
        <scope>test</scope>
    </dependency>
</dependencies>
```

Using maven - write the code

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

import org.junit.jupiter.api.Test;
import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.ValueSource;

public class NumbersTests {
        @ParameterizedTest
        @ ValueSource(ints = { 1, 3, 5, -3, 15, Integer.MAX_VALUE })       // six numbers
        void isOdd_ShouldReturnTrueForOddNumbers(int number) {
                assertTrue(Numbers.isOdd(number));
        }
}
```

```
public class Numbers {
    public static boolean isOdd(int number) {
        return number % 2 != 0;
    }
}
```

MULTIPLE VALUES

MS-Test specific

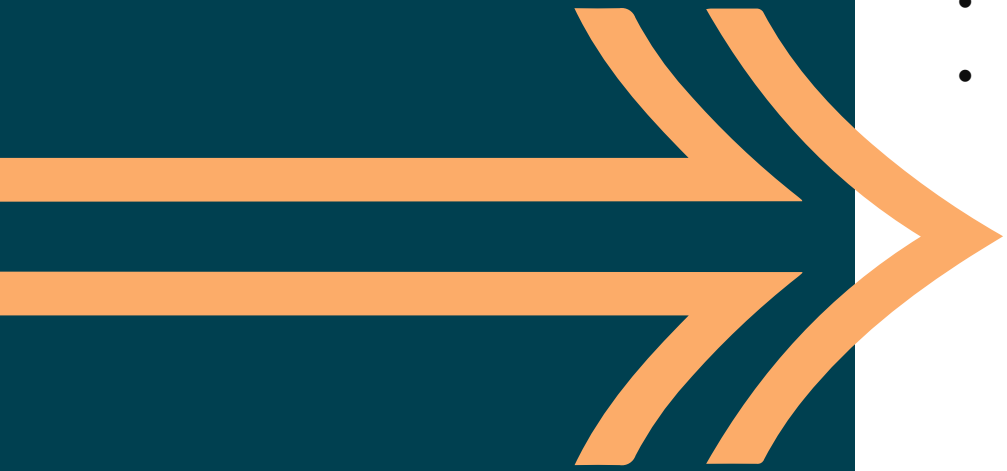
```
[DataTestMethod]
[DataRow(12, 3, 4)]
[DataRow(12, 2, 6)]
[DataRow(12, 4, 3)]
public void DivideTest(int n, int d, int q)
{
        int res = Calculator.div(n,d);
        Assert.AreEqual(q, res);
}
```

Parameterised creation method

- Hides attributes essential to fixtures, but irrelevant to the test
- Factor out fixture object creation from setUp to PCM
- Useful when creating a complex mock
- Consider also Builder Pattern

Example: Extra constructor

- If existing constructor hard-codes some dependencies
- use an extra constructor to inject dependencies by test (mock or stub)
- Introduces a ‘trap door’ to make code easier to test



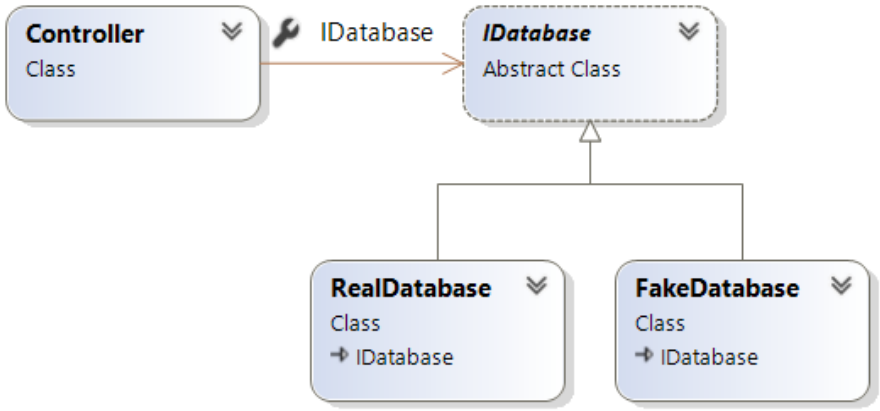
Constructor and Setter Injections

- **Constructor injection:** API signals that the parameter(s) isn't optional, you must supply it when creating the object
- **Setter injection:** API signals the dependency is optional / changeable
- Allows that to be overridden (with a mock, in the test):

```
interface IDatabase{       // or abstract
        List<Customer> getCustomers();
        // other methods
}
```

```
public class Controller {
    IDatabase db;

    public Controller(IDatabase db) {
        this.db = db;
    }
}
```




```
IDatabase db = new FakeDatabase();
Controller myController = new Controller(db);
```

Test-specific subclass

- Create a behaviour-modifying or state-exposing subclass

```
public class UserService {
    protected boolean validateAmount(double amount) {
        return amount > 0;
    }
}
```

Method to test
But can't!

```
public class TestableUserService extends UserService{

  public boolean testValidateAmount(double amount) {
        return validateAmount(amount);
  }
}
```

Test the untestable method by extension

FACTORY

Factory provides means for test to change type returned

- Test sets factory to return mock or stub
- Application code semantics completely unchanged

Use factory pattern for dependencies

- E.g. CUT uses factory method
  - 'Extract and override': extract dependency creation to factory
  - Override method in test-specific subclass

```
public class UserFactory {
    public static User createAdmin() {
        return new User("admin", "admin@QA.com");
    }

    public static User createUserWithRole(String role, String name) {
        return new User(role, name + "@QA.com");
    }
}
```

See pages 17-18 in your learner guide for an example of the factory pattern

OBJECT MOTHER

Centralizes the creation of predefined object configurations for tests.

It is like a factory but less complex often using static methods with no parameters

```
public class UserMother {
    public static User adminUser() {
        return new User("admin", "admin@QA.com", true);
    }

    public static User microsoftUser() {
        return new User("user", "bob@Microsoft.com", false);
    }
}
```

Lab

QA

Learn. To Change.