POWERING POTENTIAL

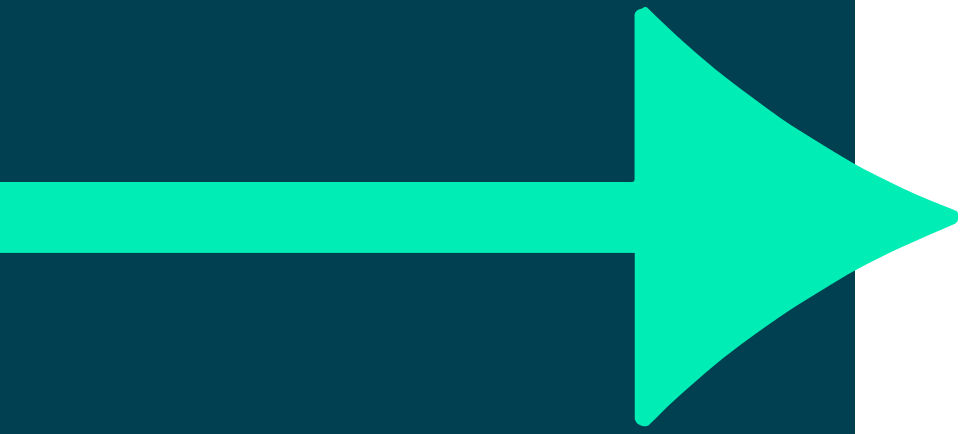
TESTING

TESTING PATTERNS

ASSERTIONS

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

TYPES OF ASSERTION



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

Guard assertion: assert a precondition for the test to be correct and follow it with the resulting state.

Delta assertion: if you can't guarantee the absolute resulting state, test the delta (difference) between the initial and resulting states. Often used for none functional testing such as performance which may not assert against a single value but a range.

Custom assertion: helps your test code respect DRY ('don't repeat yourself', i.e. there aren't any duplications).

JUNIT HANDLING EXCEPTIONS AND TIMEOUTS

(REVIEW)

- `@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 Exceptions with Junit 5 (review)

```
@Test
void testWithdrawWithInsufficientFunds() {

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

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

}
```

```
public class InsufficientFundsException extends Exception {

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

MS-Test specific

Exceptions

```
[TestMethod()]
[ExpectedException(typeof(ArgumentOutOfRangeException))]
public void TransferFundsTest()  {
        int res = Calculator.add("123", null);
}
```

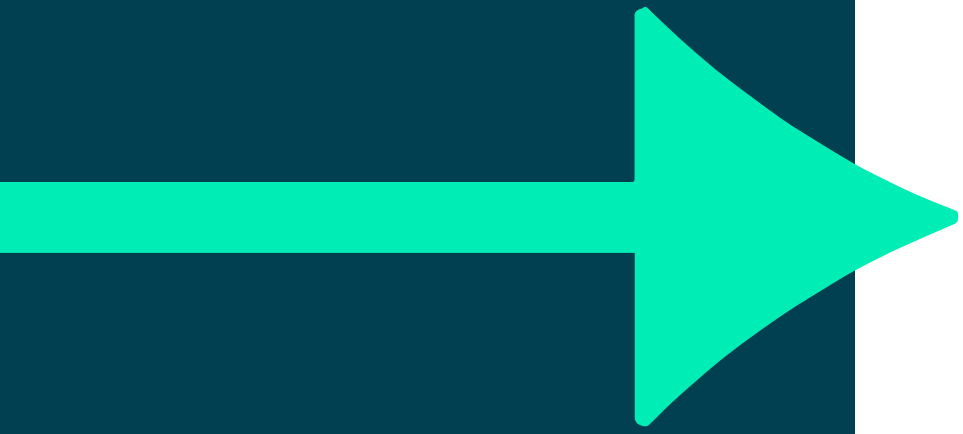
Multiple values

```
[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, esp. if it will be used in multiple tests
- Consider also Builder Pattern

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

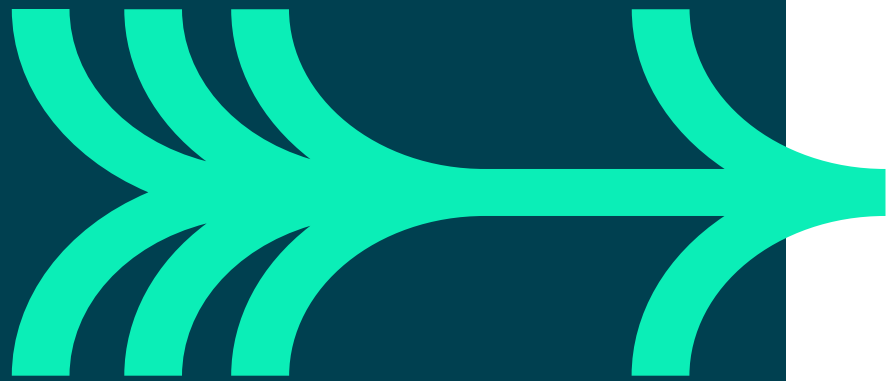
- **Constructor injection:** API signals that the parameter(s) isn't optional, you must supply it when creating the object
- **Setter injection:** API signals the dependency is optional / changeable
- With either injection, you can have a default constructor which hard-codes the default dependency, e.g. a setter which allows that to be overridden (with a mock, in the test):

```
interface IDatabase{
        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
- Not usually a TDD approach
- See lottery class below
  - Default constructor creates real **`RandomNumberGenerator`**
  - **`NumberGenerator`** field has `protected` visibility
  - **`TestableLottery`** extends `Lottery`

```
class TestableLottery extends Lottery {

    public TestableLottery(NumberGenerator generator) {
        this.generator = generator;
    }
}
```

- Inherits method to test as-is
- Test creates instance of testable subclass, injecting mock

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
  - **Or:** Client asks factory class for dependency

```
public Lottery() {

    generator = NumberGeneratorFactory.create();
}
```

See pages 17-18 of your learner guide for an example

OBJECT MOTHER

Create example objects that you can use in your tests

- Customise the objects you create
- Update the objects during the tests
- Delete object from a database once tests are completed

- Factors out creation of business objects to factory class, or just a class containing fixtures to avoid duplication
- e.g. a set of standard ‘personas’:

An example of Object Mother

```
public class HR {      // CUT
    public bool makePayment(Employee emp, double money) {
        PaymentSystem paymentSystem = new PaymentSystem();
        return paymentSystem.Pay(emp.ID, money);
    }
}
```

```
public class Employee {
    double salary;
    int age;
    List<Skill> skills;

    public Employee(double salary, int age, List<Skill> skills) {
        this.salary = salary;
        this.age = age;
    }
}
```

```
public class Skill {
    string name;
    public Skill(string skillName) {
        this.name = skillName;
    }
}
```

```
public class EmployeeObjectMother {
    public static Employee MakeEmployee(string employeeType) {
        Code to create and employee with skills, and other fields
    }
}
```

LAB

"Test Patterns" Lab

JAVA PARAMETERISED TEST

- Data-driven testing: one test method, multiple data sets
- Data hard-coded in 2D array, or from an external source

```
@Parameters
public static Collection makeData()
{
    return Arrays.asList(new Object[][] {
        {1, "Jan"},{3, "Mar"}, {12, "Dec"}
    });
}

// { input to fn, expected output }
public UtilsTest(int input, String output) {
    this.input = input;
    this.expected = output;
}
```

PARAMETERISED TEST

JUnit 4 - you can set up parameterised tests with annotations:

```
@RunWith(value= Parameterized.class)
public class UtilsTest {
    private int input;
    private String expected;
}
```

The test method uses instance variables of the class:

```
@Test
public void testGetMonthString()
{
    assertEquals("Incorrect month String", expected,
                 Utils.getMonthString(input));
}
```

Putting it all together

```
@RunWith(Parameterized.class)
public class ParameterizedTests {
    @Parameters
    public static Collection<Object[]> data() {
        return Arrays.asList(new Object[][] {
                { 0, 0 }, { 1, 1 }, { 2, 1 }, { 3, 2 }, { 4, 3 }, { 5, 5 }, { 6, 8 }
           });
    }

    private int fInput;
    private int fExpected;

    public ParameterizedTests(int input, int expected) {
        this.fInput = input;
        this.fExpected = expected;
    }

    @Test
    public void test() {
        assertEquals(fExpected, Fibonacci.compute(fInput));
    }
}
```

```
public class Fibonacci {
  public static int compute(int n) {
    int result = 0;
    if (n <= 1)
          result = n;
    else
       result = compute(n - 1) + compute(n - 2);

    return result;
    }
}
```

An alternative - Using maven

Create a maven project and then add JUNIT5 to your POM file

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">
        <modelVersion>4.0.0</modelVersion>
        <groupId>qaa</groupId>
        <artifactId>qaa</artifactId>
        <version>0.0.1-SNAPSHOT</version>
        <dependencies>
                <dependency>
                        <groupId>org.junit.jupiter</groupId>
                        <artifactId>junit-jupiter-params</artifactId>
                        <version>5.7.0</version>
                        <scope>test</scope>
                </dependency>
        </dependencies>
</project>
```

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;
    }
}
```