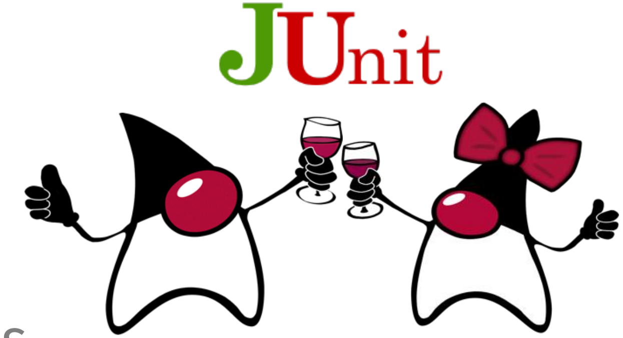
Testing with JUnit

CO7005 Software Development Techniques



Dr Stuart Cunningham
s.cunningham@chester.ac.uk

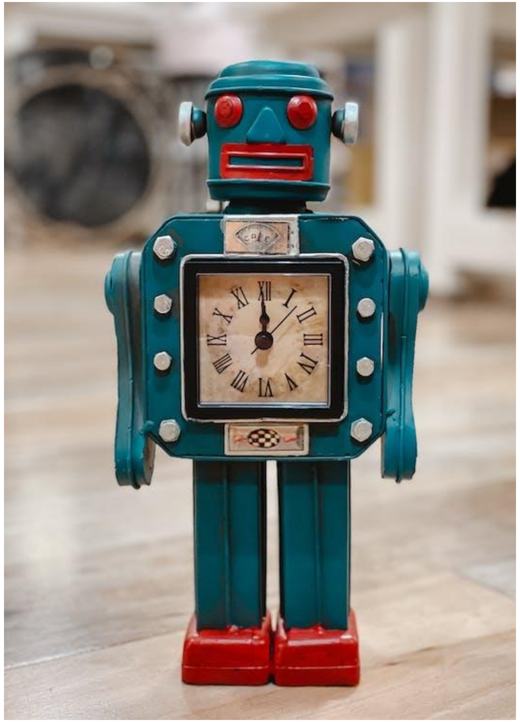
Manual Testing

- To do well, must be methodical
- Needs repeating each time code changes
- Often requires ‘test’ programs to be created
- Efficient in small programs, otherwise…
    - Time consuming
    - Not best use of developer time
    - (Human) error prone
    - Subjective
    - Expensive

Manual Testing

- Remember the **CestrianInsurance** program?
- Three variables, each tested for two states
  - (car | motorcycle); (age: < 25 | ≥ 25); (points: ≤ 6 | > 6)
  - Minimum **8 tests** to ensure functionality
- Imagine we add *scooter* to vehicles and an *over 50* age bracket)
  - (car | motorcycle | scooter); (age: < 25 | ≥ 25 | > 50); (points: ≤ 6 | > 6)
  - Now at least **18 tests** are needed
- Its *combinatorics* make it exhausting

Automated Testing



- A special program that interrogates the program in development (to be tested)
  - E.g. using JUnit (https://junit.org) framework
- Requires test cases/definitions
- Defined tests can be run many times
- Helps find errors and bugs
- *Much* faster than manual testing
- Efficient in large programs or projects

Automated Tests

- Unit Testing → Regression Testing
  - Focus is small elements of functionality (*units*)
  - In Java, typically testing per *method*
  - Core focus if adopting *Test-driven Development*

We're focusing here

- Integration Testing
  - How units and larger components work *together*
  - In Java, may be between methods, classes, interfaces, packages, etc.
- System Testing
  - Functionality of *entire system* being developed
  - Determining if the whole application or program meets *requirements*

Defining Unit Testing

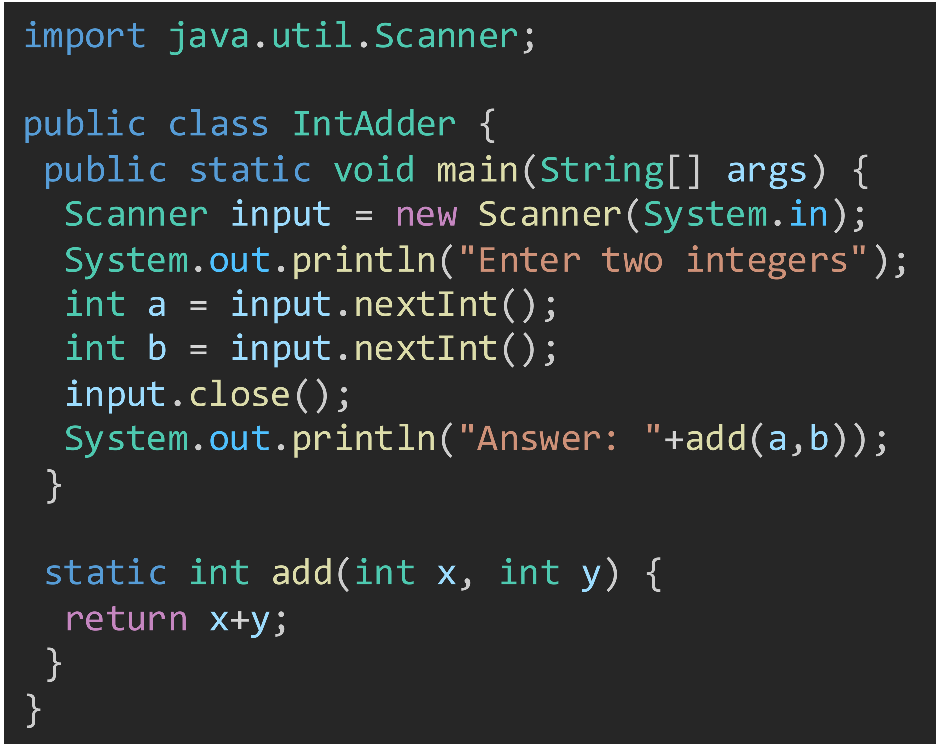
*“A test is an assessment of our knowledge, a proof of concept, or an examination of data. A class test is an examination of our knowledge to ascertain whether we can go to the next level. **For software, it is the validation of functional and nonfunctional requirements before it is shipped to a customer**.”*

(Acharya 2014)

JUnit Tests

- Integration of JUnit varies by IDE, but once setup it follows common principles
- Tests are Java programs (`*.java`) and (`*.class`) files
- Utilise classes, interfaces and methods from the framework
- Makes use of @Annotations and multiple Assertions
- Can do a lot with `@Test` and `assertEquals()` alone

An Addition Program



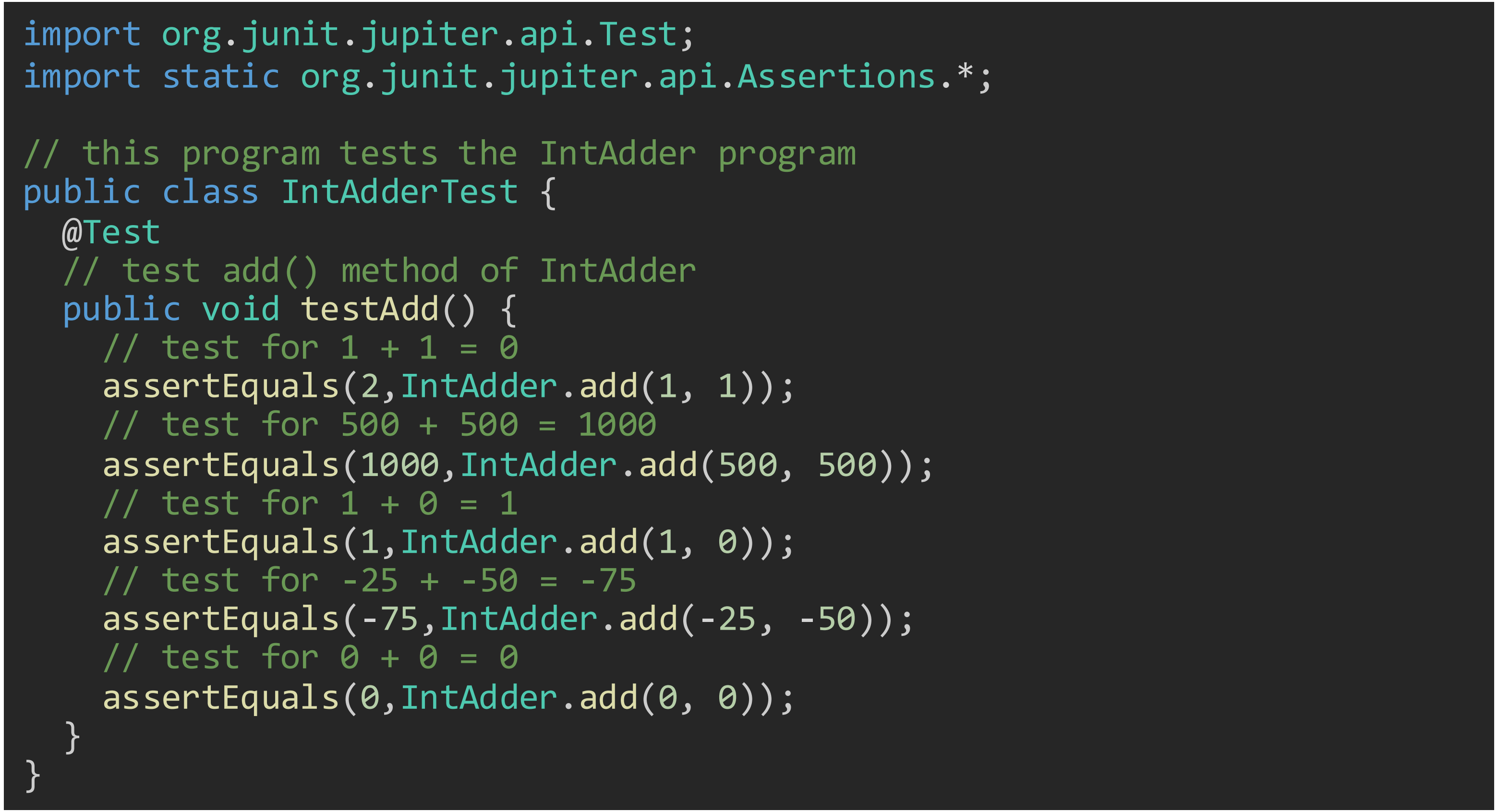
```java
import java.util.Scanner;

public class IntAdder {
 public static void main(String[] args) {
  Scanner input = new Scanner(System.in);
  System.out.println("Enter two integers");
  int a = input.nextInt();
  int b = input.nextInt();
  input.close();
  System.out.println("Answer: "+add(a,b));
 }

 static int add(int x, int y) {
  return x+y;
 }
}
```

- Define inputs and expected outputs
- Check a variety of possible values (e.g. zero and negative numbers)
- For example:
  - 1 + 1 = 2
  - 500 + 500 = 1000
  - 1 + 0 = 1
  - -25 + -50 = -75
  - 0 + 0 = 0

Testing an Addition Program – Single

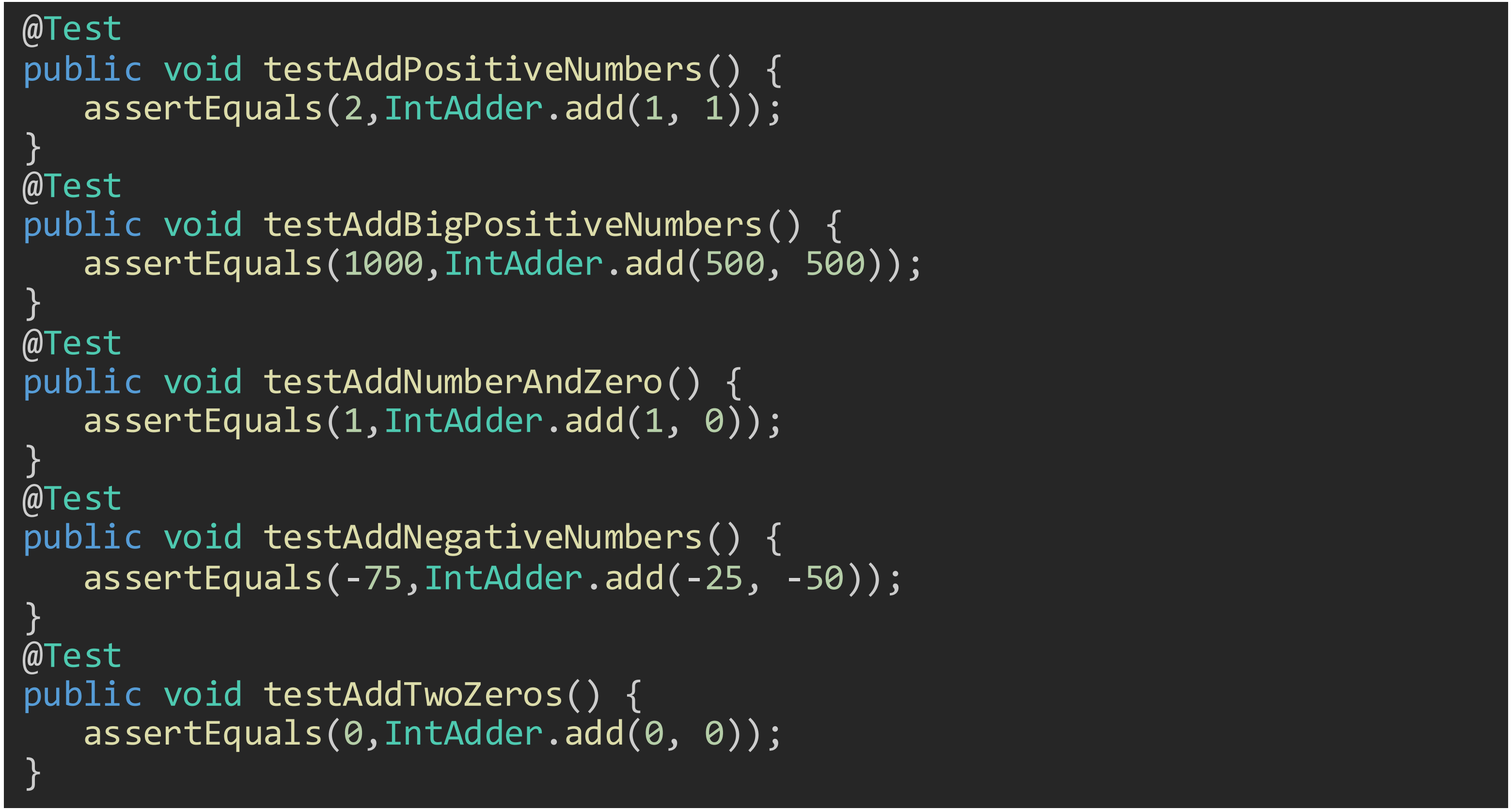
```java
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;


// this program tests the IntAdder program
public class IntAdderTest {
  @Test
  // test add() method of IntAdder
  public void testAdd() {
    // test for 1 + 1 = 0
    assertEquals(2,IntAdder.add(1, 1));
  }
}
```

Testing an Addition Program – Multiple



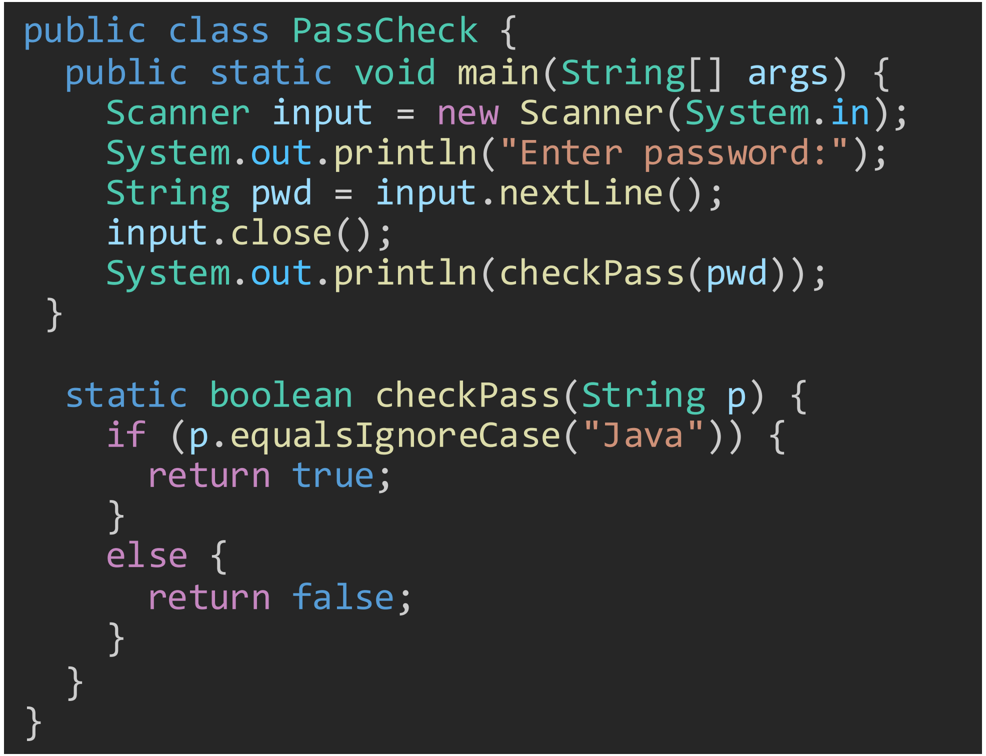
```
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;

// this program tests the IntAdder program
public class IntAdderTest {
  @Test
  // test add() method of IntAdder
  public void testAdd() {
    // test for 1 + 1 = 0
    assertEquals(2,IntAdder.add(1, 1));
    // test for 500 + 500 = 1000
    assertEquals(1000,IntAdder.add(500, 500));
    // test for 1 + 0 = 1
    assertEquals(1,IntAdder.add(1, 0));
    // test for -25 + -50 = -75
    assertEquals(-75,IntAdder.add(-25, -50));
    // test for 0 + 0 = 0
    assertEquals(0,IntAdder.add(0, 0));
  }
}
```

Testing an Addition Program – Better



```java
@Test
public void testAddPositiveNumbers() {
   assertEquals(2,IntAdder.add(1, 1));
}
@Test
public void testAddBigPositiveNumbers() {
   assertEquals(1000,IntAdder.add(500, 500));
}
@Test
public void testAddNumberAndZero() {
   assertEquals(1,IntAdder.add(1, 0));
}
@Test
public void testAddNegativeNumbers() {
   assertEquals(-75,IntAdder.add(-25, -50));
}
@Test
public void testAddTwoZeros() {
   assertEquals(0,IntAdder.add(0, 0));
}
```

A Password Program



```
public class PassCheck {
 public static void main(String[] args) {
  Scanner input = new Scanner(System.in);
  System.out.println("Enter password:");
  String pwd = input.nextLine();
  input.close();
  System.out.println(checkPass(pwd));
 }

 static boolean checkPass(String p) {
  if (p.equalsIgnoreCase("Java")) {
   return true;
  }
  else {
   return false;
  }
 }
}
```

- Consider valid and invalid inputs and responses
- Return type is Boolean
  - `Java = true`
  - `java = true`
  - `JAVA = true`
  - `Pascal = false`
  - `C = false`
  - `hypertext = false`

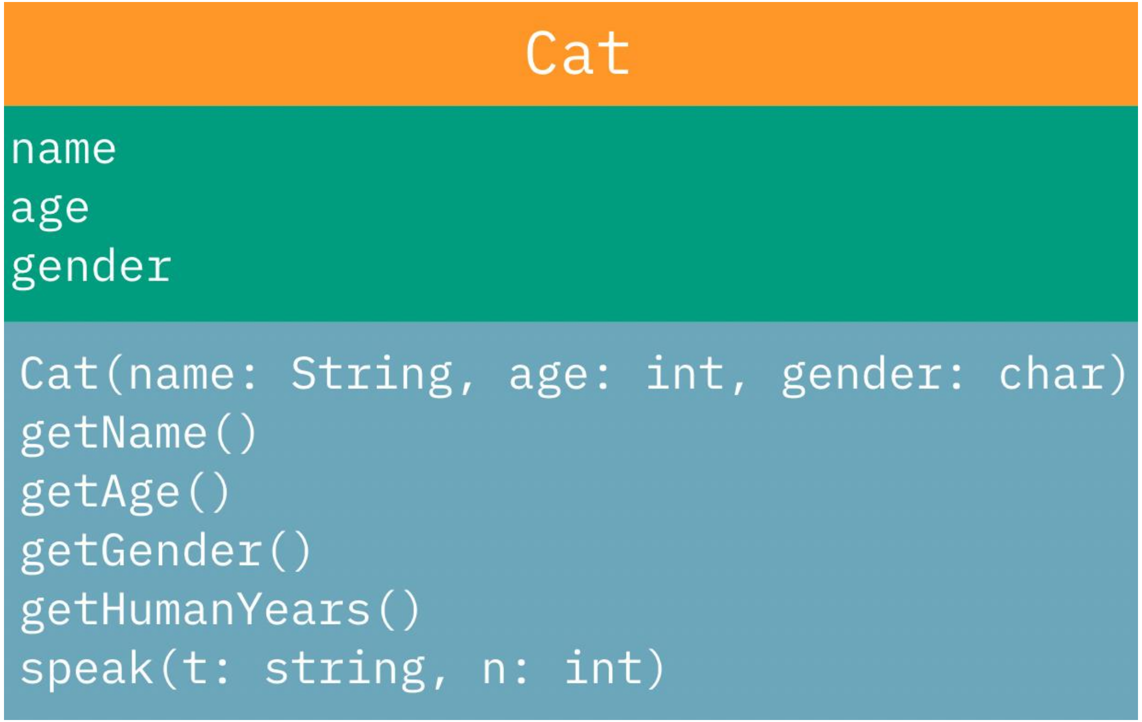
Testing a Password Program

```java
public class PassCheckTest {
 // valid entries
 @Test
 public void testCheckPassValidCamelCase(){
    assertTrue(PassCheck.checkPass("Java"));
 }
 @Test
 public void testCheckPassValidUpperCase(){
    assertTrue(PassCheck.checkPass("JAVA"));
 }
 @Test
 public void testCheckPassValidLowerCase(){
    assertTrue(PassCheck.checkPass("java"));
 }
```

```java
 // invalid entries
 @Test
 public void testCheckPassInvalidCamelCase() {
    assertFalse(PassCheck.checkPass("Pascal"));
 }
 @Test
 public void testCheckPassInvalidUpperCase() {
    assertFalse(PassCheck.checkPass("C"));
 }
 @Test
 public void testCheckPassInvalidLowerCase() {
    assertFalse(PassCheck.checkPass("hypertext"));
 }
}
```

Testing a Class (Cat.java)

- Various methods, including constructor
- Must create object instance(s) for testing
- Check for expected values



Cat

name
age
gender

Cat(name: String, age: int, gender: char)
getName()
getAge()
getGender()
getHumanYears()
speak(t: string, n: int)

VAT.java

```
>> java VAT
Enter value to calculate cost (inc.VAT): 10
Cost: £12.00
```

```java
import java.text.NumberFormat;
import java.util.Scanner;

class VAT {
  public static void main(String[] args) {
    Scanner input = new Scanner(System.in);
    System.out.print("Enter value to calculate cost (inc.VAT): ");
    double cost = calcIncVAT(input.nextDouble());
    input.close();
    // output value in currency format
    System.out.println("Cost: "+NumberFormat.getCurrencyInstance().format(cost));
  }
  static double calcIncVAT(double val) {
    if (val <=0) {
      throw new ArithmeticException("Zero or less");
    }
    else {
      final double rate = 0.2;
      return (val*rate)+val;
    }
  }
}
```

VAT.java - Exception Testing

- Check if exception is thrown under prescribed condition
- Can also check the error message is correct / expected

```
@Test
@DisplayName("Arithmetic Exception <=0 input")
public void testCalcIncVATArithmeticException() {
  Exception error;
  error = assertThrows(ArithmeticException.class, () -> VAT.calcIncVAT((0)));
  assertEquals("Zero or less", error.getMessage());
}
```

- Second parameter is a Lambda Expression

VAT.java - Parameterized Testing

- Used to test a series of values
  - Specified here with `@ValueSource`
  - Note definition of plural variable type (i.e. `doubles`)
- Performs the same test many times with different values

```
@ParameterizedTest
@ValueSource(doubles = {1, 5, 10, 15, 20, 25, 30})
@DisplayName("Param Test: Many Values")
public void testCalcIncVatManyVals(double costs) {
  assertEquals(costs*1.2, VAT.calcIncVAT(costs));
}
```

References

Acharya, S. (2014). *Mastering Unit Testing Using Mockito and JUnit* (1st ed.). Packt Publishing.

JUnit 5 User Guide: https://junit.org/junit5/docs/current/user-guide/