Ollscoil na Gaillimhe
University of Galway

CT326 Programming III

LECTURE 5
UNIT TESTING IN JAVA &
TEST DRIVEN DEVELOPMENT
PART 2

- DR. ADRIAN CLEAR -
SCHOOL OF COMPUTER SCIENCE

Objectives

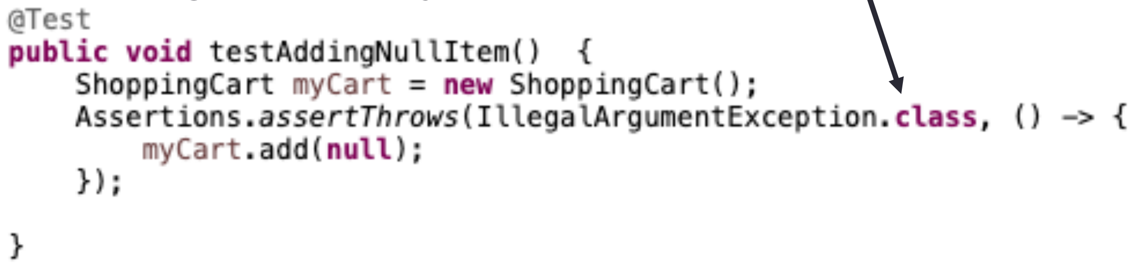
- Learn how to test for exceptions
- Understand how to test code with dependencies

Testing exceptions

- We can use the assertThrows assertion to test if an exception has been thrown
- Specifying our assumption is that an IllegalArgumentException will be thrown if we try to add a null Item to the cart

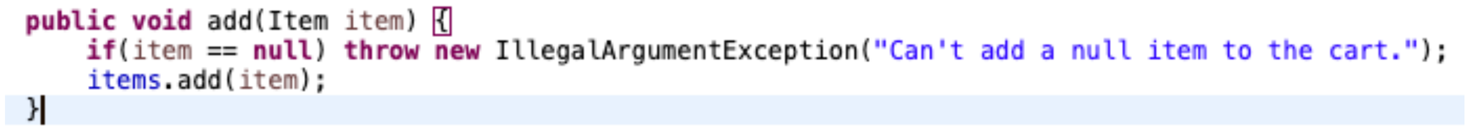
Don't forget the '.class'

ShoppingCartTests.java



```
@Test
public void testAddingNullItem()  {
    ShoppingCart myCart = new ShoppingCart();
    Assertions.assertThrows(IllegalArgumentException.class, () -> {
        myCart.add(null);
    });

}
```

ShoppingCart.java



```
public void add(Item item) {
    if(item == null) throw new IllegalArgumentException("Can't add a null item to the cart.");
    items.add(item);
}
```

What tests should we write?

- Test single units of functionality in isolation
- Not integration tests
- Multiple tests for a single piece of logic (multiple scenarios)
- Each test will cover a single scenario for a single piece of logic

Unit testing techniques

- Equivalence testing
  - possible inputs are partitioned into equivalence classes, and a test case is selected for each class
  - minimises number of test cases
  - systems usually behave in similar ways for all members of a class
- Boundary testing
  - special case of equivalence testing that focuses on the conditions at the boundary of the equivalence classes
  - boundaries often overlooked by developers
- Path testing
  - by exercising all possible paths through the code at least once, most faults will trigger failures
  - requires knowledge of the source code and data structures

Equivalence classes example

- Suppose customers can register for our online shop and we we want a method to test whether a mobile phone number that they enter is valid.
- Equivalence classes (valid)
  - 10 digit number that begins with 083 (Test case: 0833456789)
  - 10 digit number that begins with 085 (Test case: 0853456789)
  - 10 digit number that begins with 086 (Test case: 0863456789)
  - 10 digit number that begins with 087 (Test case: 0873456789)
  - 10 digit number that begins with 089 (Test case: 0893456789)
- Equivalence classes (invalid)
  - an input that is not a number (Test case: ABC)
  - a <10 digit number (Test case: 55)
  - a >10 digit number (Test case: 123456789101112)
  - a 10 digit number that doesn’t begin with 083, 085, 086, 087, or 089 (Test case: 0123456789)

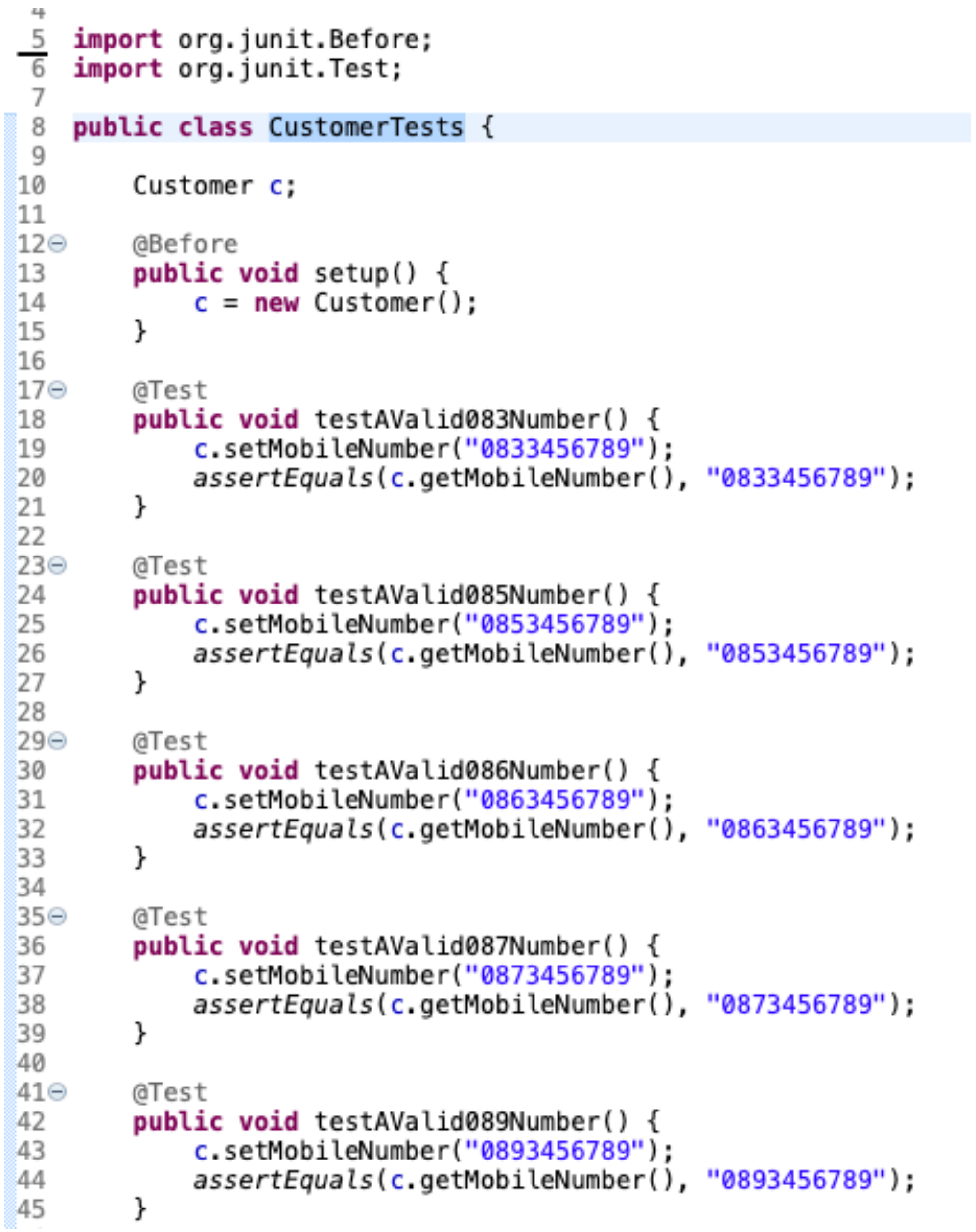
Boundary tests

- Focuses on the conditions at the boundary of the equivalence classes
- Instead of selecting any element in the equivalence class, boundary testing requires that the elements be selected from the “edges” of the equivalence class
- Assumption is that developers often overlook special cases at the boundary of the equivalence classes
- Boundary cases
  - a **10-digit** input that is not a number (Test case: ABCDEFGHIJ)
  - a **10-digit** input that begins with 083, 085, 086, 087, or 089 but is not a number (Test case: 087DEFGHIJ)
  - a 9-digit number that begins with 083, 085, 086, 087, or 089 (Test case: 086123456)
  - an 11-digit number that begins with 083, 085, 086, 087, or 089 (Test case: 08612345678)

Exercise

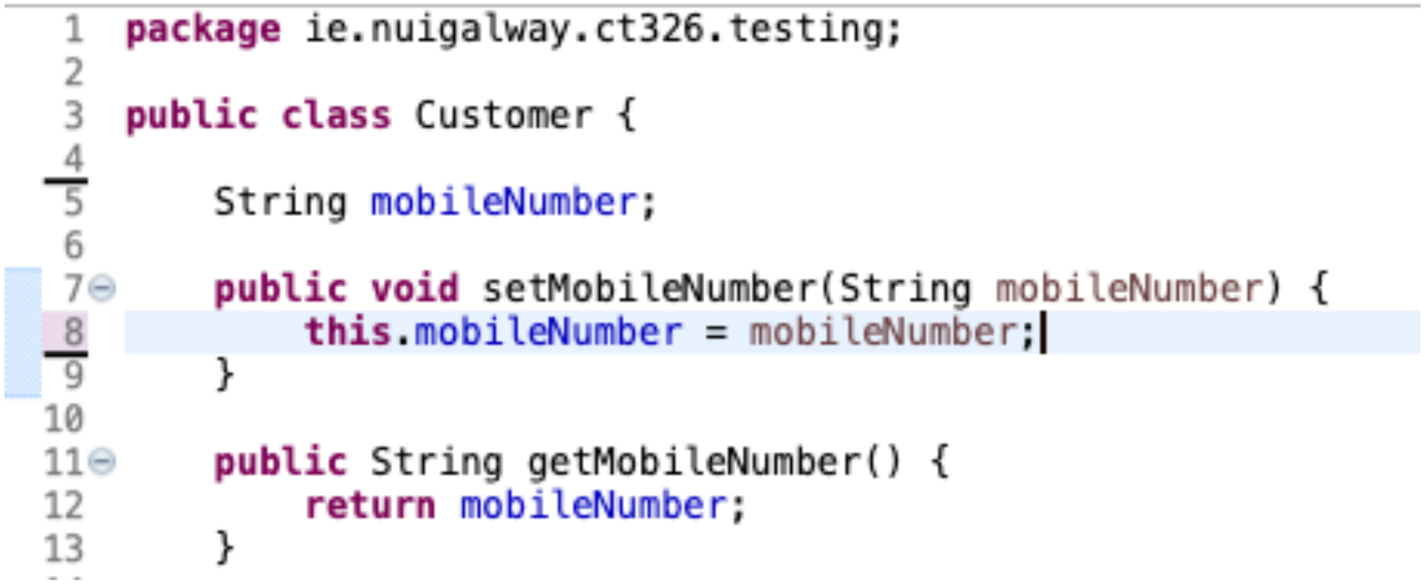
- Use a TDD approach to write a method in a Customer class to add a valid mobile phone number

Valid cases



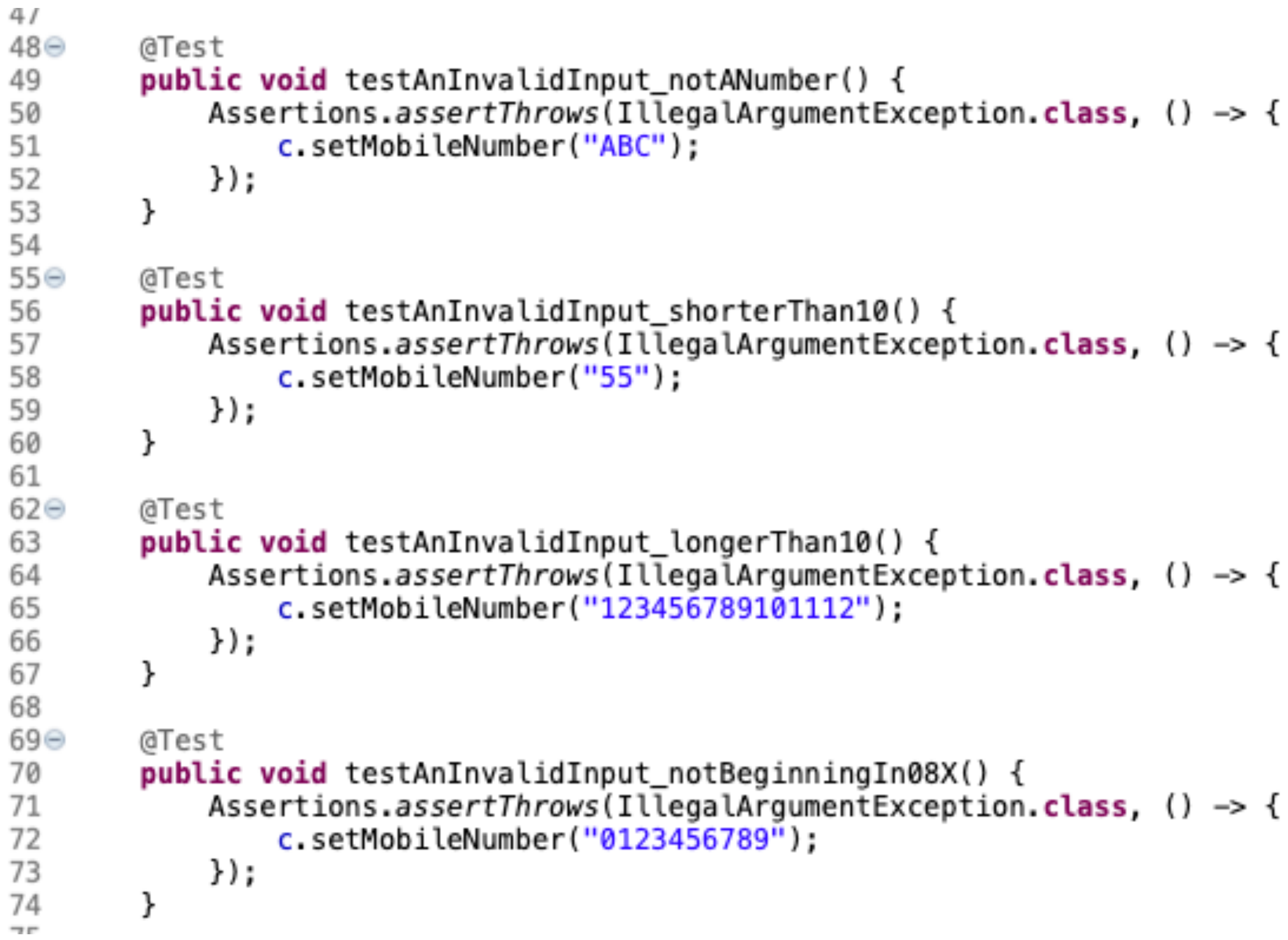
```java
import org.junit.Before;
import org.junit.Test;

public class CustomerTests {

    Customer c;

    @Before
    public void setup() {
        c = new Customer();
    }

    @Test
    public void testAValid083Number() {
        c.setMobileNumber("0833456789");
        assertEquals(c.getMobileNumber(), "0833456789");
    }

    @Test
    public void testAValid085Number() {
        c.setMobileNumber("0853456789");
        assertEquals(c.getMobileNumber(), "0853456789");
    }

    @Test
    public void testAValid086Number() {
        c.setMobileNumber("0863456789");
        assertEquals(c.getMobileNumber(), "0863456789");
    }

    @Test
    public void testAValid087Number() {
        c.setMobileNumber("0873456789");
        assertEquals(c.getMobileNumber(), "0873456789");
    }

    @Test
    public void testAValid089Number() {
        c.setMobileNumber("0893456789");
        assertEquals(c.getMobileNumber(), "0893456789");
    }
```

Ollscoil na Gaillimhe
University of Galway

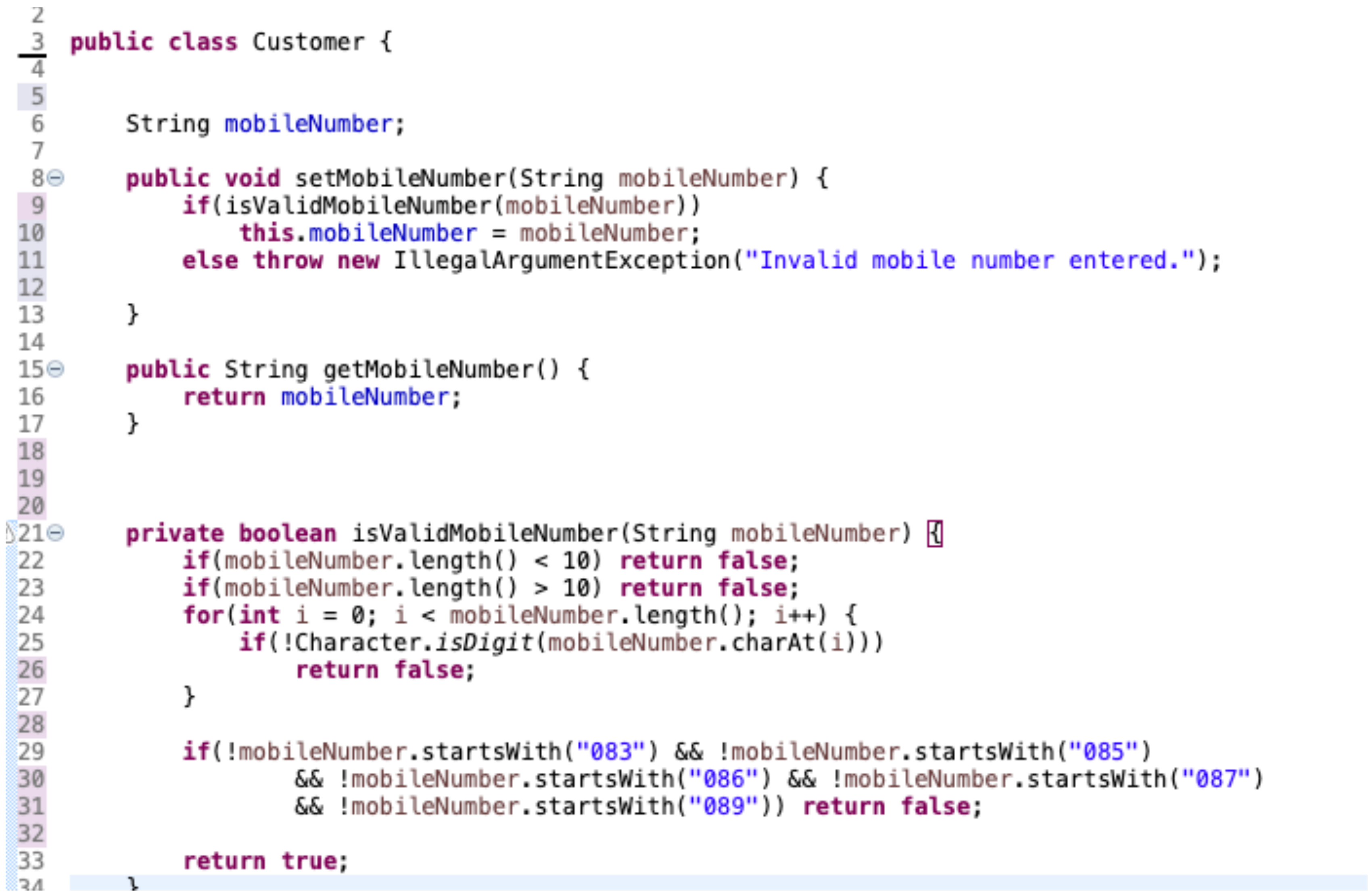


```
package ie.nuigalway.ct326.testing;

public class Customer {

    String mobileNumber;

    public void setMobileNumber(String mobileNumber) {
        this.mobileNumber = mobileNumber;
    }

    public String getMobileNumber() {
        return mobileNumber;
    }
```

Invalid cases

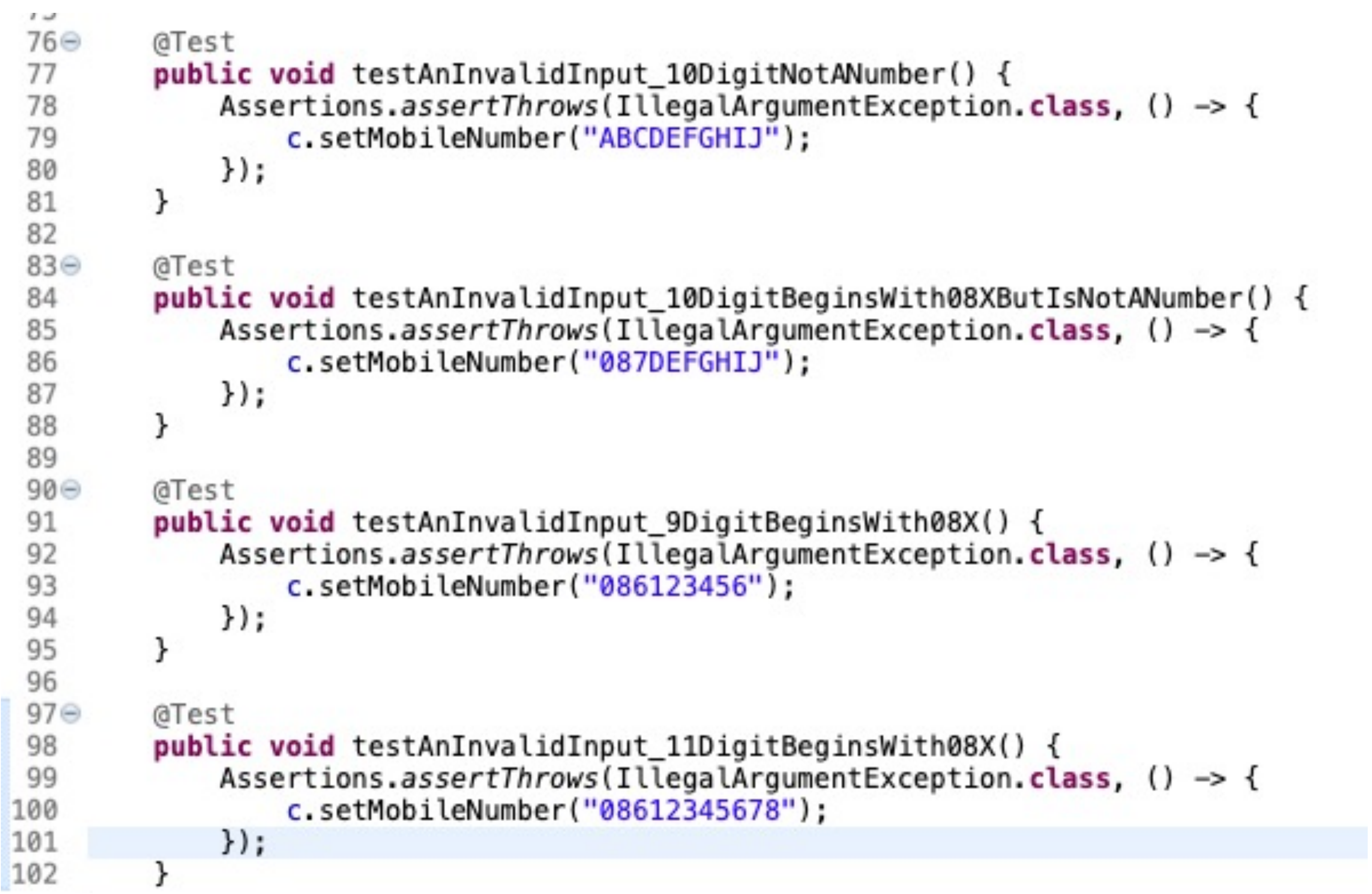


```java
    @Test
    public void testAnInvalidInput_notANumber() {
        Assertions.assertThrows(IllegalArgumentException.class, () -> {
            c.setMobileNumber("ABC");
        });
    }

    @Test
    public void testAnInvalidInput_shorterThan10() {
        Assertions.assertThrows(IllegalArgumentException.class, () -> {
            c.setMobileNumber("55");
        });
    }

    @Test
    public void testAnInvalidInput_longerThan10() {
        Assertions.assertThrows(IllegalArgumentException.class, () -> {
            c.setMobileNumber("123456789101112");
        });
    }

    @Test
    public void testAnInvalidInput_notBeginningIn08X() {
        Assertions.assertThrows(IllegalArgumentException.class, () -> {
            c.setMobileNumber("0123456789");
        });
    }
```



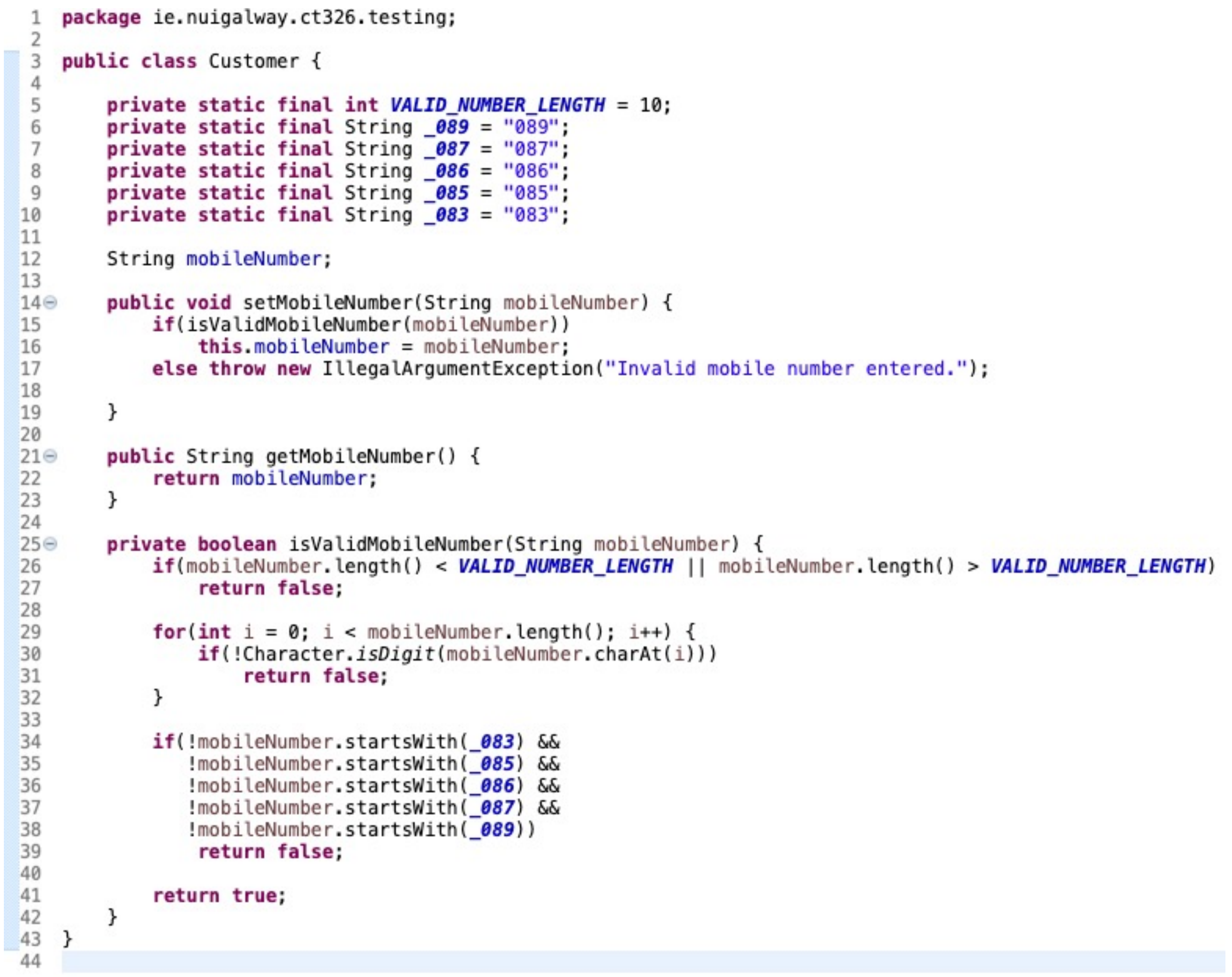
```java

public class Customer {


    String mobileNumber;

    public void setMobileNumber(String mobileNumber) {
        if(isValidMobileNumber(mobileNumber))
            this.mobileNumber = mobileNumber;
        else throw new IllegalArgumentException("Invalid mobile number entered.");

    }

    public String getMobileNumber() {
        return mobileNumber;
    }



    private boolean isValidMobileNumber(String mobileNumber) {
        if(mobileNumber.length() < 10) return false;
        if(mobileNumber.length() > 10) return false;
        for(int i = 0; i < mobileNumber.length(); i++) {
            if(!Character.isDigit(mobileNumber.charAt(i)))
                return false;
        }

        if(!mobileNumber.startsWith("083") && !mobileNumber.startsWith("085")
                && !mobileNumber.startsWith("086") && !mobileNumber.startsWith("087")
                && !mobileNumber.startsWith("089")) return false;

        return true;
    }
```

Boundary cases



```
    @Test
    public void testAnInvalidInput_10DigitNotANumber() {
        Assertions.assertThrows(IllegalArgumentException.class, () -> {
            c.setMobileNumber("ABCDEFGHIJ");
        });
    }

    @Test
    public void testAnInvalidInput_10DigitBeginsWith08XButIsNotANumber() {
        Assertions.assertThrows(IllegalArgumentException.class, () -> {
            c.setMobileNumber("087DEFGHIJ");
        });
    }

    @Test
    public void testAnInvalidInput_9DigitBeginsWith08X() {
        Assertions.assertThrows(IllegalArgumentException.class, () -> {
            c.setMobileNumber("086123456");
        });
    }

    @Test
    public void testAnInvalidInput_11DigitBeginsWith08X() {
        Assertions.assertThrows(IllegalArgumentException.class, () -> {
            c.setMobileNumber("08612345678");
        });
    }
```

Refactor



```java
package ie.nuigalway.ct326.testing;

public class Customer {

    private static final int VALID_NUMBER_LENGTH = 10;
    private static final String _089 = "089";
    private static final String _087 = "087";
    private static final String _086 = "086";
    private static final String _085 = "085";
    private static final String _083 = "083";

    String mobileNumber;

    public void setMobileNumber(String mobileNumber) {
        if(isValidMobileNumber(mobileNumber))
            this.mobileNumber = mobileNumber;
        else throw new IllegalArgumentException("Invalid mobile number entered.");

    }

    public String getMobileNumber() {
        return mobileNumber;
    }

    private boolean isValidMobileNumber(String mobileNumber) {
        if(mobileNumber.length() < VALID_NUMBER_LENGTH || mobileNumber.length() > VALID_NUMBER_LENGTH)
            return false;

        for(int i = 0; i < mobileNumber.length(); i++) {
            if(!Character.isDigit(mobileNumber.charAt(i)))
                return false;
        }

        if(!mobileNumber.startsWith(_083) &&
           !mobileNumber.startsWith(_085) &&
           !mobileNumber.startsWith(_086) &&
           !mobileNumber.startsWith(_087) &&
           !mobileNumber.startsWith(_089))
            return false;

        return true;
    }
}
```

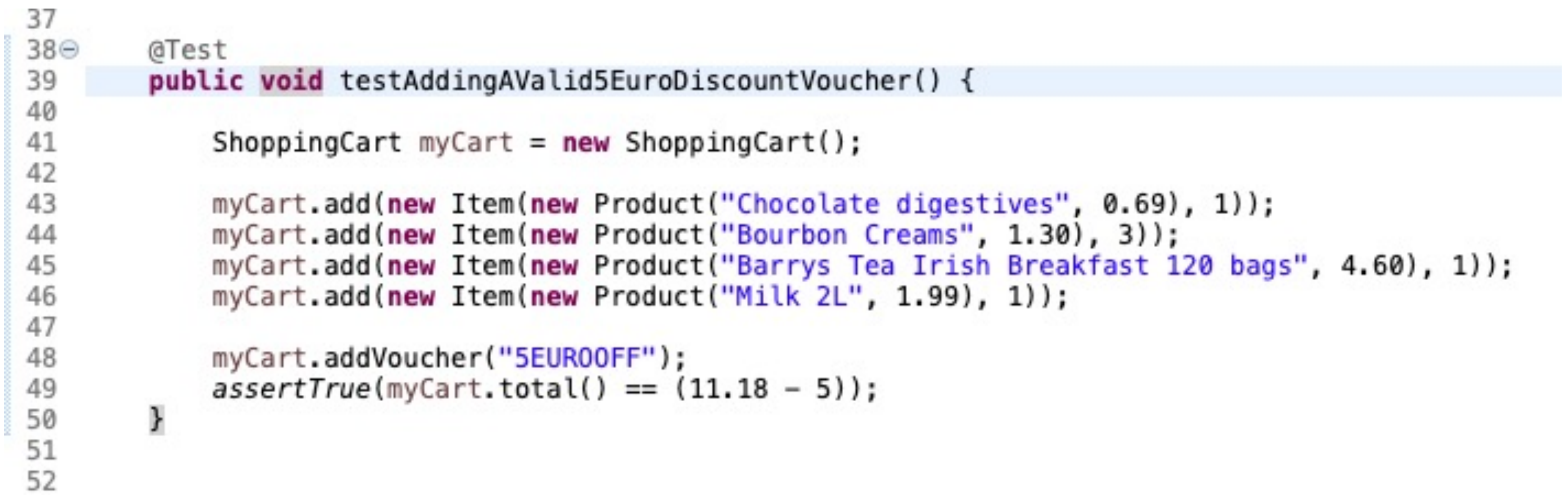
Testing code with dependencies

- Often the application logic that we want to test will have some dependencies on external services or components.
- In unit testing, we want to isolate our component under test from any dependencies
  - otherwise we're doing **integration testing**
- This is problematic as our application logic won't work without its dependencies
- **Solution:** We can create a **stub** to simulate the functionality of this external component

Stub example: discount vouchers

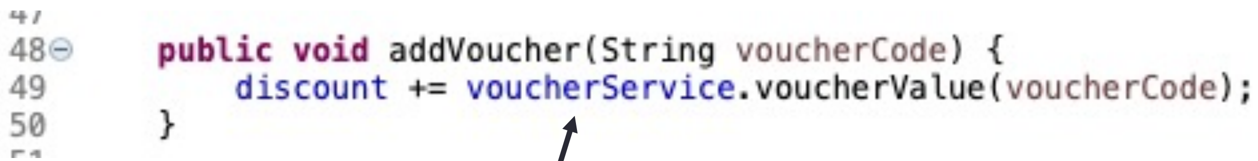
- Suppose we have functionality to add a voucher to our shopping cart which can result in a monetary discount
- However, the validation of vouchers is done by an external web service which returns the value of the voucher to be discounted from the shopping cart total
- We want to test that when we add a valid voucher, we get the correct total for our shopping cart

Our test case…



```
    @Test
    public void testAddingAValid5EuroDiscountVoucher() {

        ShoppingCart myCart = new ShoppingCart();

        myCart.add(new Item(new Product("Chocolate digestives", 0.69), 1));
        myCart.add(new Item(new Product("Bourbon Creams", 1.30), 3));
        myCart.add(new Item(new Product("Barrys Tea Irish Breakfast 120 bags", 4.60), 1));
        myCart.add(new Item(new Product("Milk 2L", 1.99), 1));

        myCart.addVoucher("5EUROOFF");
        assertTrue(myCart.total() == (11.18 - 5));
    }
```

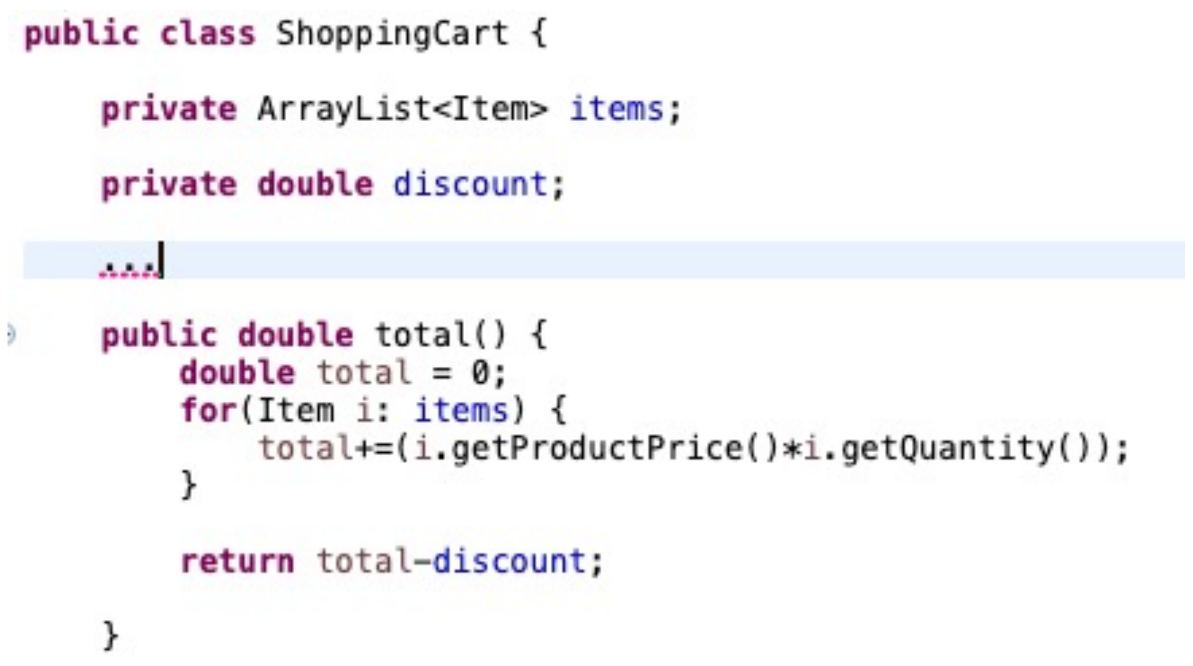
…and our implementation…



```
    public void addVoucher(String voucherCode) {
        discount += voucherService.voucherValue(voucherCode);
    }
```

we haven't implemented this yet

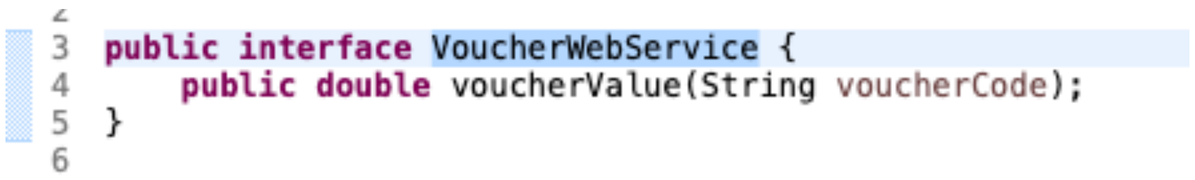
- We are going to use a **discount** variable to keep track of our total to discount
- We can then change our existing **total()** method to subtract the discount before returning the total cost of the cart.



```
public class ShoppingCart {

    private ArrayList<Item> items;

    private double discount;

    ...

    public double total() {
        double total = 0;
        for(Item i: items) {
            total+=(i.getProductPrice()*i.getQuantity());
        }

        return total-discount;

    }
```

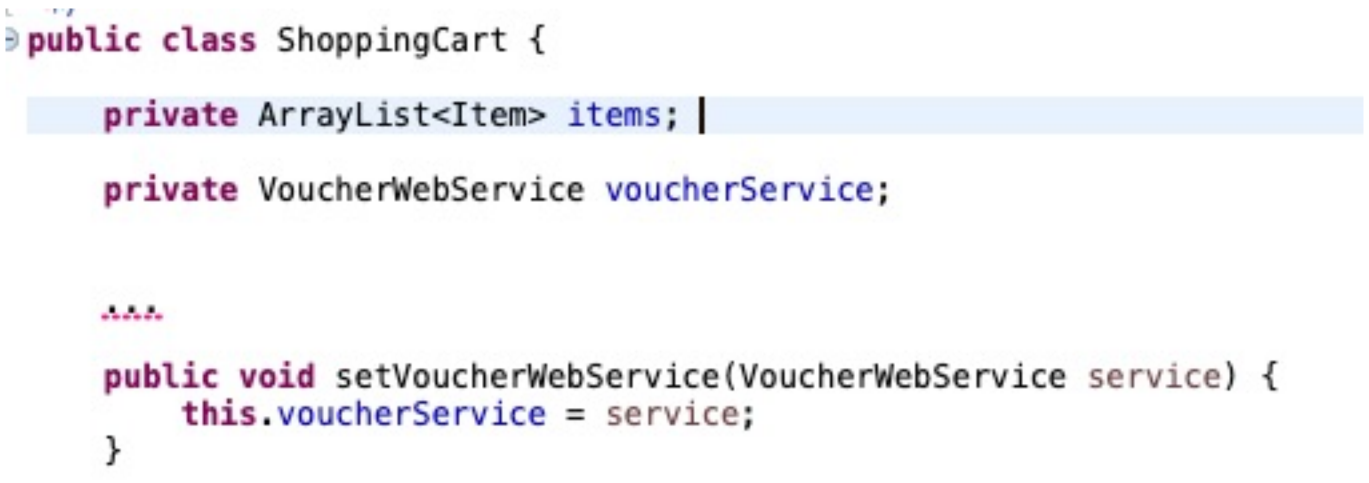
Our stub…

- Create an interface to represent the external service



```java
public interface VoucherWebService {
    public double voucherValue(String voucherCode);
}
```

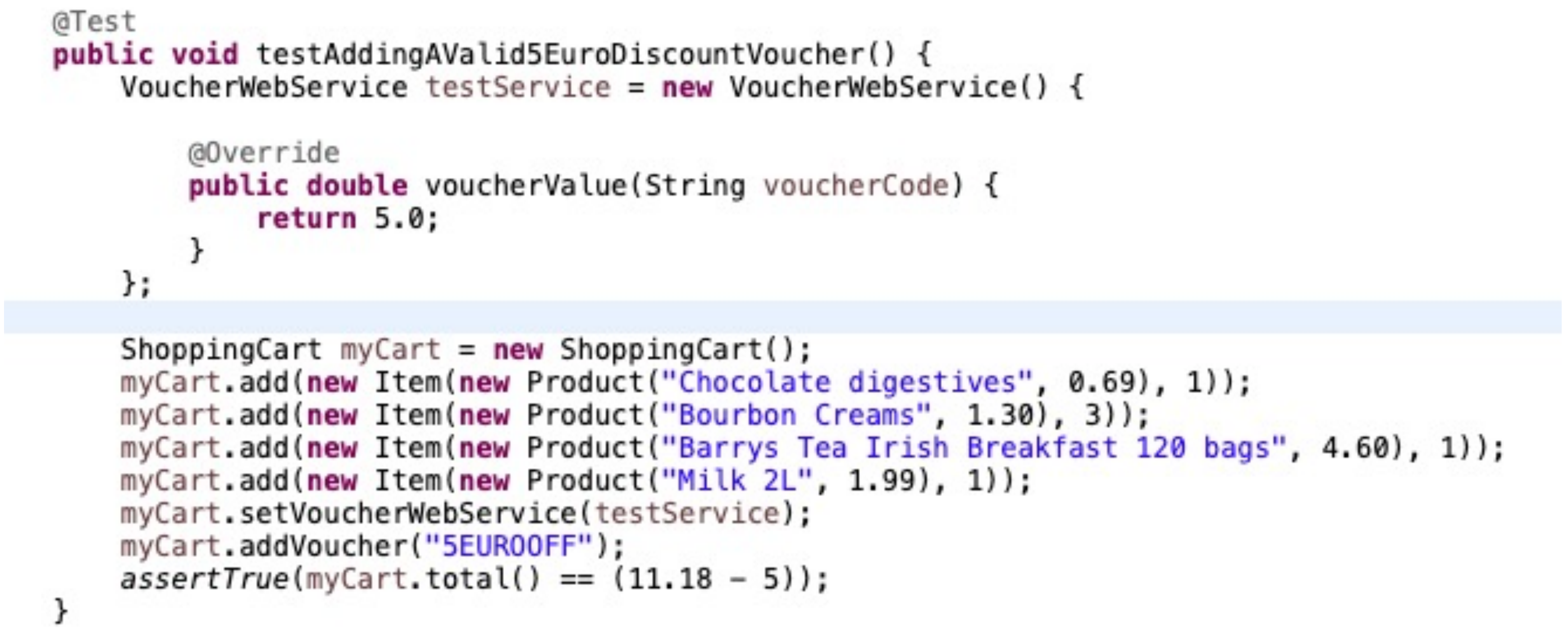
Now, let's use it in our application



```
public class ShoppingCart {

    private ArrayList<Item> items;

    private VoucherWebService voucherService;


    ...

    public void setVoucherWebService(VoucherWebService service) {
        this.voucherService = service;
    }
```

OLLSCOIL NA GAILLIMHE
UNIVERSITY OF GALWAY

…and inject it into our test



```java
@Test
public void testAddingAValid5EuroDiscountVoucher() {
    VoucherWebService testService = new VoucherWebService() {

        @Override
        public double voucherValue(String voucherCode) {
            return 5.0;
        }
    };

    ShoppingCart myCart = new ShoppingCart();
    myCart.add(new Item(new Product("Chocolate digestives", 0.69), 1));
    myCart.add(new Item(new Product("Bourbon Creams", 1.30), 3));
    myCart.add(new Item(new Product("Barrys Tea Irish Breakfast 120 bags", 4.60), 1));
    myCart.add(new Item(new Product("Milk 2L", 1.99), 1));
    myCart.setVoucherWebService(testService);
    myCart.addVoucher("5EUROOFF");
    assertTrue(myCart.total() == (11.18 - 5));
}
```

TDD guidelines

- Test the expected outcome of an example
- Think about examples and outcomes, not code or how it should work in detail
- Don’t pre-judge design… let your tests drive it
- Write the minimum code to get your tests to pass
- Each test should validate one single piece of logic

Coverage and Path testing

- **Code coverage** is a measure of how many lines of your code are executed by automated tests
- **Path testing** refers to test cases that exercise all possible paths through the code at least once
  - idea is that most faults will trigger failures in this way
- Requires knowledge of the source code and data structures
- Impractical to achieve 100% code coverage for large projects

Summary

- **Test Driven Development** is an iterative software development process where the production of tests drive the development of the code
  - Consists of a cycle of Red, Green, Refactor
- **Unit testing** finds differences between a specification of an object and its realisation as a component
- Unit testing in TDD involves the production of **test cases** which are sets of inputs and expected outcomes for examples of use of a test component
- The purpose of test cases in TDD is to cause failures and detect faults that point to missing or erroneous implementation of specified functionality

Next time…

- Strings