

Test Driven Development

More JUnit Tests for the Shop app

Produced Dr. Siobhán Drohan
by: Mairead Meagher
 Siobhan Roche

Topic List

- Product and ProductTest.java
- JUnit Testing of Store.java
- Testing Driver.java

Product.java

```
package models;

import utils.Utilities;

public class Product {

    private String productName = "";
    private int productCode = -1;
    private double unitCost = 0;
    private boolean inCurrentProductLine = false;

    public Product(String productName, int productCode, double unitCost, boolean inCurrentProductLine) {
        this.productName = Utilities.truncateString(productName, 20);
        setProductCode(productCode);
        this.unitCost = unitCost;
        this.inCurrentProductLine = inCurrentProductLine;
    }

    public String toString() {
        return "Product description: " + productName
            + ", product code: " + productCode
            + ", unit cost: " + unitCost
            + ", currently in product line: " + Utilities.booleanToYN(inCurrentProductLine);
    }
}
```

```
public void setProductCode(int productCode) {
    if (Utilities.validRange(productCode, 1000, 9999)) {
        this.productCode = productCode;
    }
}

public void setProductName(String productName) {
    if (Utilities.validateStringLength(productName, 20)) {
        this.productName = productName;
    }
}

public void setUnitCost(double unitCost) {
    this.unitCost = unitCost;
}

public void setInCurrentProductLine(boolean inCurrentProductLine) {
    this.inCurrentProductLine = inCurrentProductLine;
}

public String getProductName(){
    return productName;
}

public double getUnitCost(){
    return unitCost;
}

public int getProductCode() {
    return productCode;
}

public boolean isInCurrentProductLine() {
    return inCurrentProductLine;
}
```

Product.java

```
Structure
```

```
Product
```

- m g Product(String, int, double, boolean)
- m g getName(): String
- m g getUnitCost(): double
- m g getCode(): int
- m g isInCurrentProductLine(): boolean
- m g setCode(int): void
- m g setName(String): void
- m g setUnitCost(double): void
- m g setInCurrentProductLine(boolean): void
- m g toString(): String ↳ Object
- f 🔒 productName: String = ""
- f 🔒 productCode: int = -1
- f 🔒 unitCost: double = 0
- f 🔒 inCurrentProductLine: boolean = false



ProductTest.java

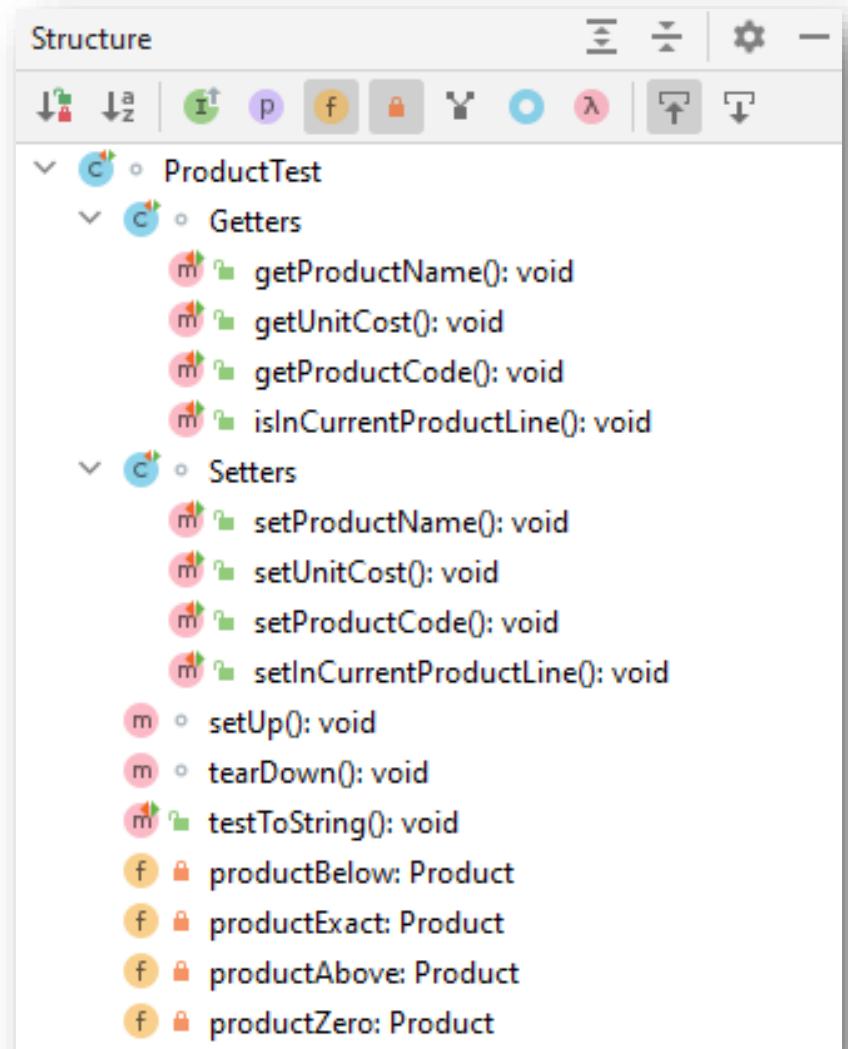
```
Structure
```

```
ProductTest
```

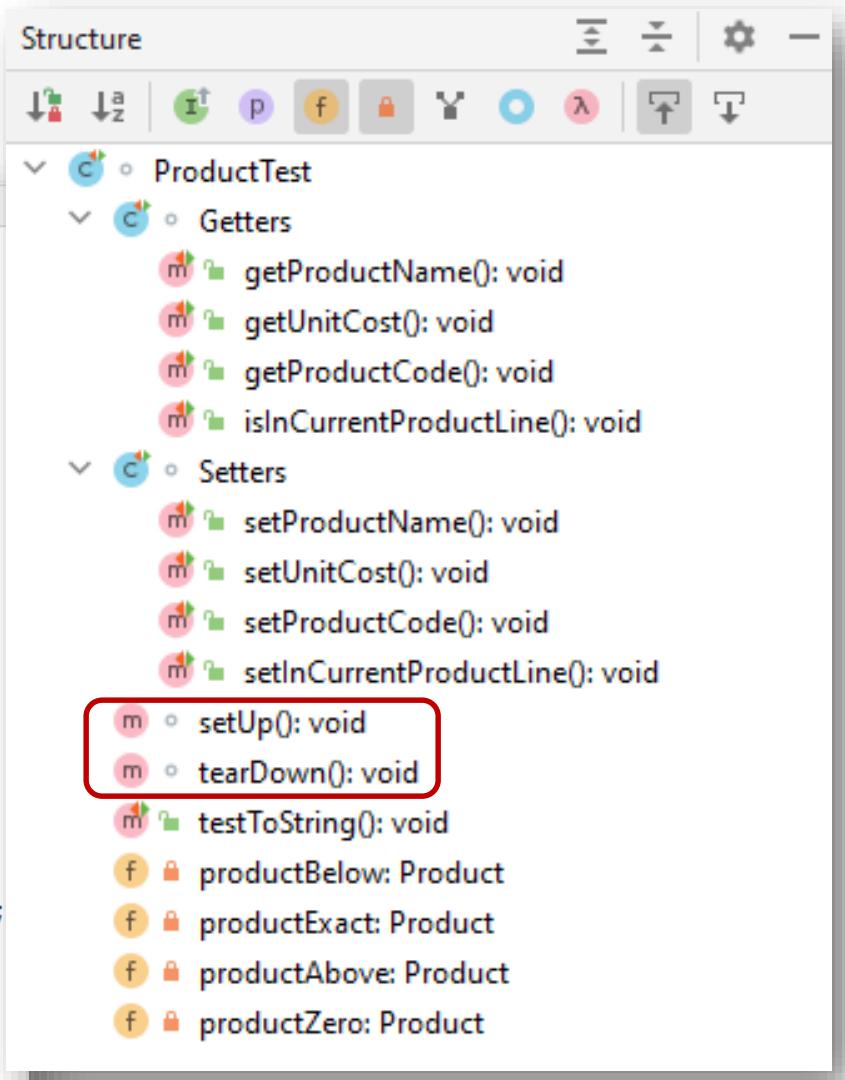
- c ↳ Getters
 - m g getName(): void
 - m g getUnitCost(): void
 - m g getCode(): void
 - m g isInCurrentProductLine(): void
- c ↳ Setters
 - m g setName(): void
 - m g setUnitCost(): void
 - m g setCode(): void
 - m g setInCurrentProductLine(): void
- m o setUp(): void
- m o tearDown(): void
- m g testToString(): void
- f 🔒 productBelow: Product
- f 🔒 productExact: Product
- f 🔒 productAbove: Product
- f 🔒 productZero: Product

ProductTest.java

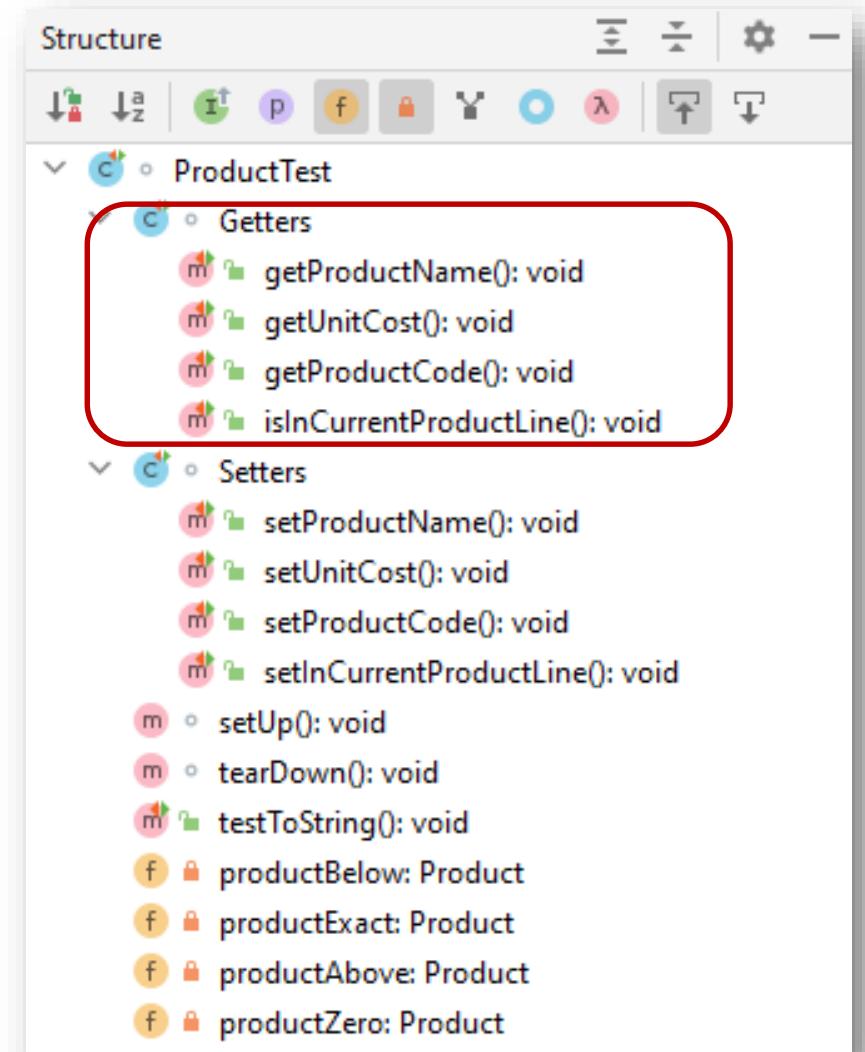
```
ProductTest.java × Product.java ×
1 package models;
2
3 import org.junit.jupiter.api.AfterEach;
4 import org.junit.jupiter.api.BeforeEach;
5 import org.junit.jupiter.api.Nested;
6 import org.junit.jupiter.api.Test;
7
8 import static org.junit.jupiter.api.Assertions.*;
9
10 class ProductTest {
11
12     private Product productBelow, productExact, productAbove, productZero;
13
14     @BeforeEach
15     void setUp() {...}
16
17     @AfterEach
18     void tearDown() { productBelow = productExact = productAbove = productZero = null; }
19
20     @Nested
21     class Getters {...}
22
23     @Nested
24     class Setters {...}
25
26     @Test
27     void testToString() {...}
28 }
```



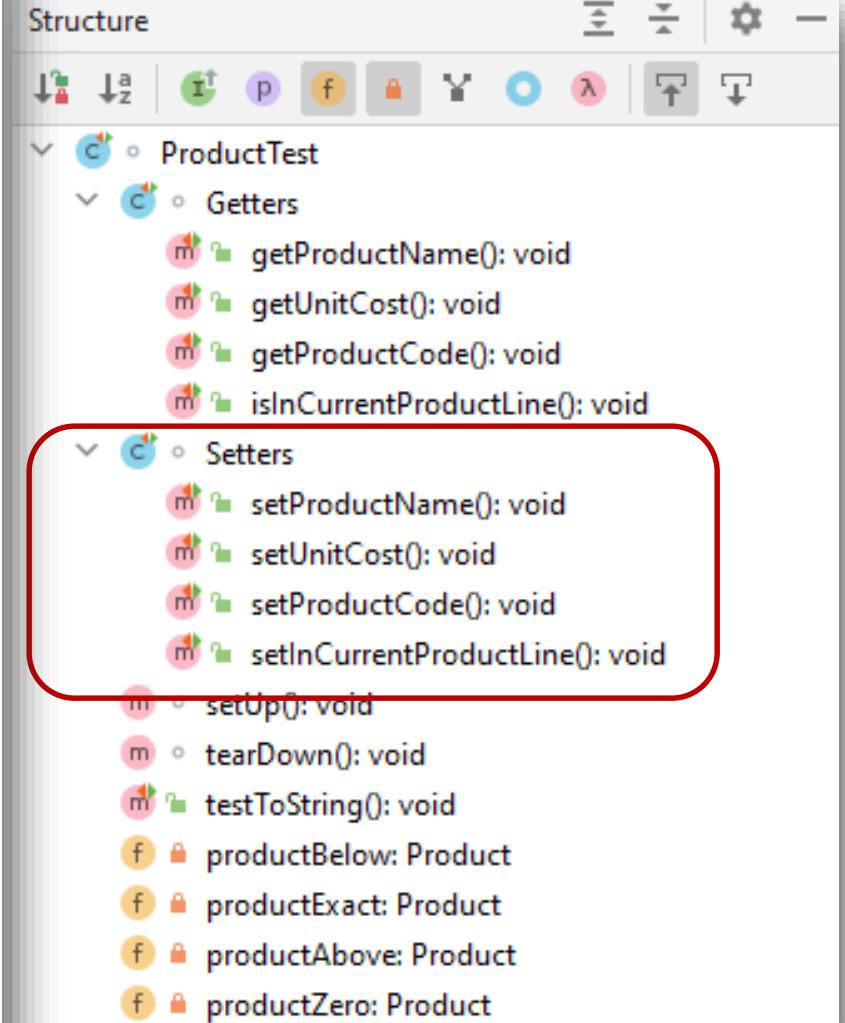
```
ProductTest.java
1 package models;
2
3 import org.junit.jupiter.api.AfterEach;
4 import org.junit.jupiter.api.BeforeEach;
5 import org.junit.jupiter.api.Nested;
6 import org.junit.jupiter.api.Test;
7
8 import static org.junit.jupiter.api.Assertions.*;
9
10 class ProductTest {
11
12     private Product productBelow, productExact, productAbove, productZero;
13
14     @BeforeEach
15     void setUp() {
16         //name, 19 chars, code 999, unitCost 1, inCurrentProductLine true.
17         productBelow = new Product( productName: "Television 42Inches", productCode: 999, unitCost: 1, inCurrentProductLine: true);
18         //name, 20 chars, code 1000, unitCost 999, inCurrentProductLine true.
19         productExact = new Product( productName: "Television 50 Inches", productCode: 1000, unitCost: 999, inCurrentProductLine: true);
20         //name, 21 chars, code 10000, unitCost 1000, inCurrentProductLine false.
21         productAbove = new Product( productName: "Television 60 Inches.", productCode: 10000, unitCost: 1000, inCurrentProductLine: false);
22         //name, 0 chars, code 9999, unitCost 0, inCurrentProductLine false.
23         productZero = new Product( productName: "", productCode: 9999, unitCost: 0, inCurrentProductLine: false);
24     }
25
26     @AfterEach
27     void tearDown() {
28         productBelow = productExact = productAbove = productZero = null;
29     }
30 }
```



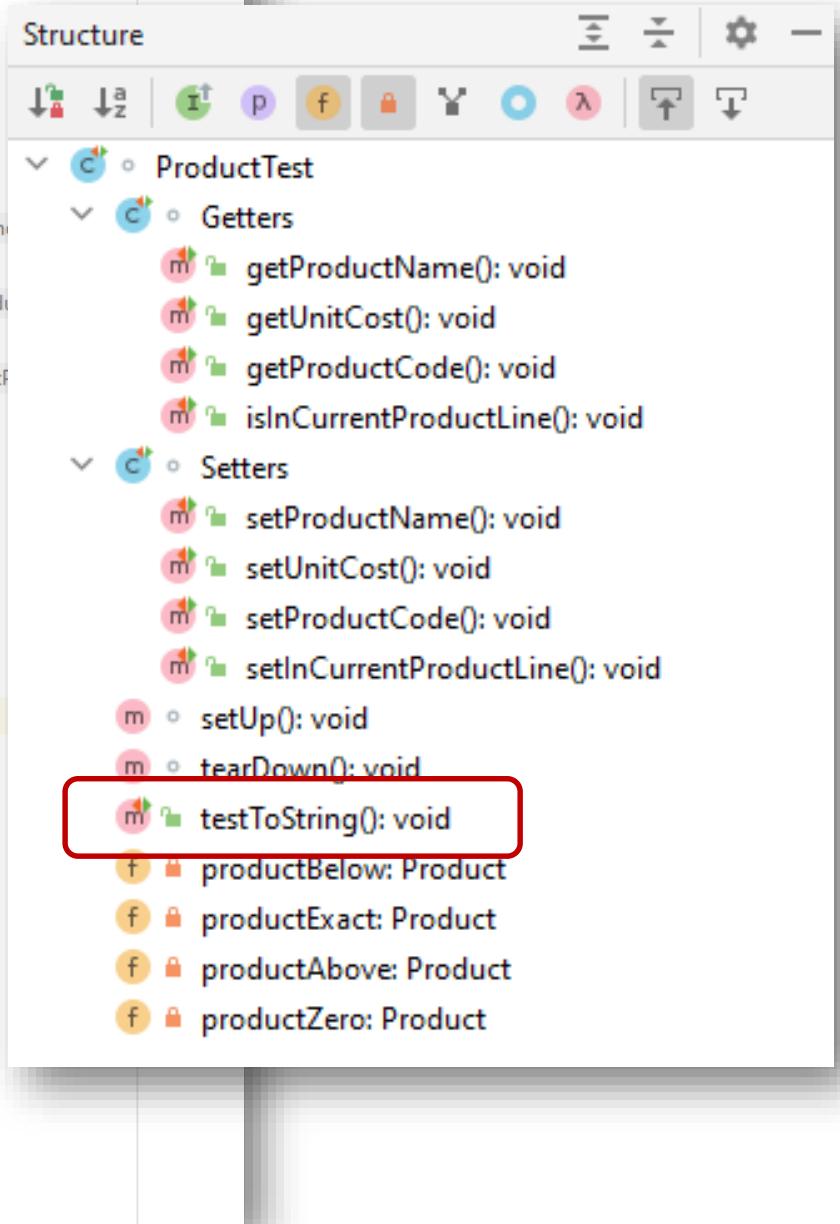
```
ProductTest.java ×  
30  
31     @Nested  
32     class Getters {  
33  
34         @Test  
35         void getProductName() {  
36             assertEquals( expected: "Television 42Inches", productBelow.getProductName());  
37             assertEquals( expected: "Television 50 Inches", productExact.getProductName());  
38             assertEquals( expected: "Television 60 Inches", productAbove.getProductName());  
39             assertEquals( expected: "", productZero.getProductName());  
40         }  
41  
42         @Test  
43         void getUnitCost() {  
44             assertEquals( expected: 1, productBelow.getUnitCost());  
45             assertEquals( expected: 999, productExact.getUnitCost());  
46             assertEquals( expected: 1000, productAbove.getUnitCost());  
47             assertEquals( expected: 0, productZero.getUnitCost());  
48         }  
49  
50         @Test  
51         void getProductCode() {  
52             assertEquals( expected: -1, productBelow.getProductCode());  
53             assertEquals( expected: 1000, productExact.getProductCode());  
54             assertEquals( expected: -1, productAbove.getProductCode());  
55             assertEquals( expected: 9999, productZero.getProductCode());  
56         }  
57  
58         @Test  
59         void isInCurrentProductLine() {  
60             assertTrue(productBelow.isInCurrentProductLine());  
61             assertTrue(productExact.isInCurrentProductLine());  
62             assertFalse(productAbove.isInCurrentProductLine());  
63             assertFalse(productZero.isInCurrentProductLine());  
64         }  
65     }  
}
```



```
ProductTest.java x
66
67     @Nested
68     class Setters {
69         @Test
70         void setProductName() {
71             assertEquals( expected: "Television 42Inches", productBelow.getProductName());
72
73             productBelow.setProductName("iPhone 13 Charcoal."); //19 chars - update performed
74             assertEquals( expected: "iPhone 13 Charcoal.", productBelow.getProductName());
75
76             productBelow.setProductName("iPhone 12 - Charcoal"); //20 chars - update performed
77             assertEquals( expected: "iPhone 12 - Charcoal", productBelow.getProductName());
78
79             productBelow.setProductName("iPhone 11: - Charcoal"); //21 chars - update ignored
80             assertEquals( expected: "iPhone 12 - Charcoal", productBelow.getProductName());
81         }
82
83         @Test
84         void setUnitCost() {
85             assertEquals( expected: 999, productExact.getUnitCost());
86             productExact.setUnitCost(99.99); //no validation performed
87             assertEquals( expected: 99.99, productExact.getUnitCost());
88         }
89
90         @Test
91         void setProductCode() {...}
92
93         @Test
94         void setInCurrentProductLine() {...}
95     }
96
97
98 }
```



```
c ProductTest.java x
9
10 class ProductTest {
11
12     private Product productBelow, productExact, productAbove, productZero;
13
14     @BeforeEach
15     void setUp() {
16         //name, 19 chars, code 999, unitCost 1, inCurrentProductLine true.
17         productBelow = new Product( productName: "Television 42Inches", productCode: 999, unitCost: 1, inCurrentProductLine: true );
18         //name, 20 chars, code 1000, unitCost 999, inCurrentProductLine true.
19         productExact = new Product( productName: "Television 50 Inches", productCode: 1000, unitCost: 999, inCurrentProductLine: true );
20         //name, 21 chars, code 10000, unitCost 1000, inCurrentProductLine false.
21         productAbove = new Product( productName: "Television 60 Inches.", productCode: 10000, unitCost: 1000, inCurrentProductLine: false );
22         //name, 0 chars, code 9999, unitCost 0, inCurrentProductLine false.
23         productZero = new Product( productName: "", productCode: 9999, unitCost: 0, inCurrentProductLine: false );
24     }
25
26     @AfterEach
27     void tearDown() { productBelow = productExact = productAbove = productZero = null; }
28
29     @Nested
30     class Getters {...}
31
32     @Nested
33     class Setters {...}
34
35     @Test
36     void testToString() {
37         String toStringContents = productExact.toString();
38         assertTrue(toStringContents.contains("Product description: " + productExact.getProductName()));
39         assertTrue(toStringContents.contains("product code: " + productExact.getProductCode()));
40         assertTrue(toStringContents.contains("unit cost: " + productExact.getUnitCost()));
41         assertTrue(toStringContents.contains(" currently in product line: Y"));
42     }
43 }
```



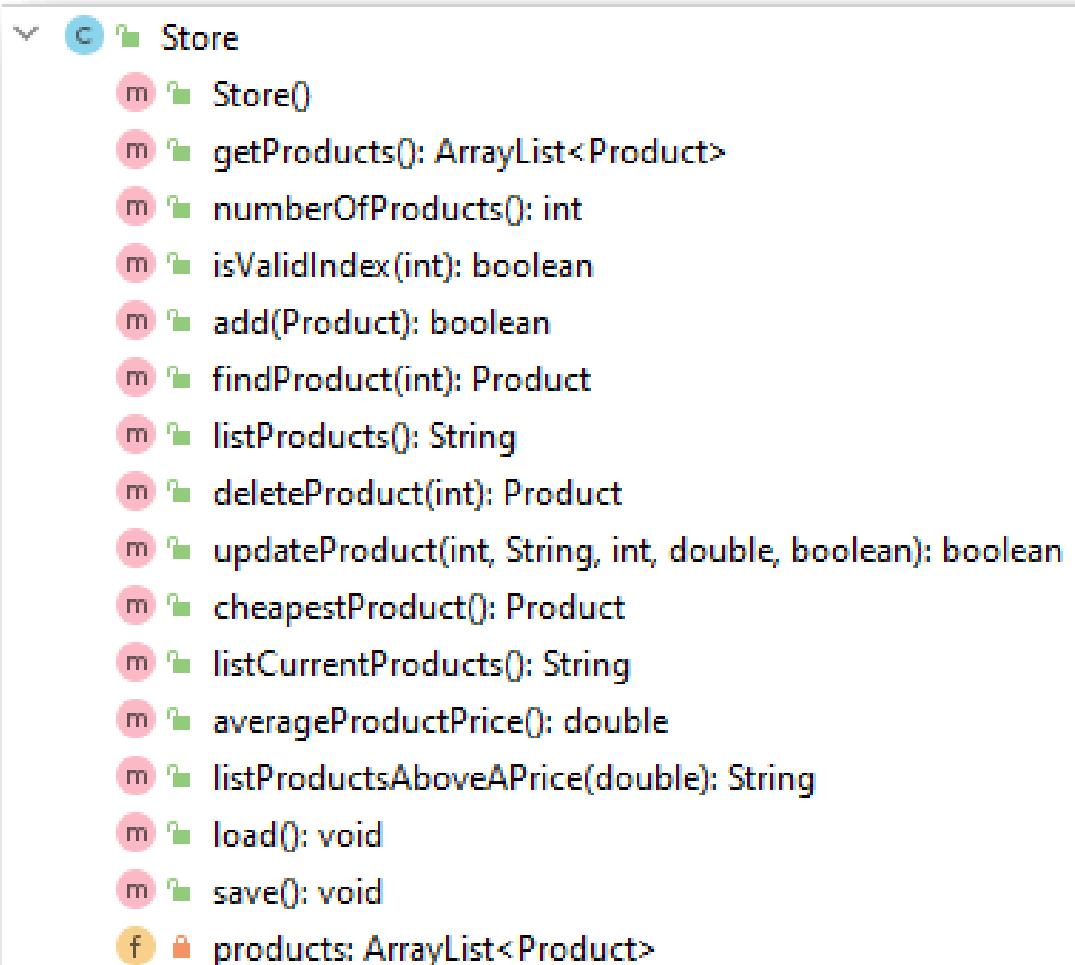
Topic List

- Product and ProductTest.java

- JUnit Testing of Store.java

- Testing Driver.java

Store.java



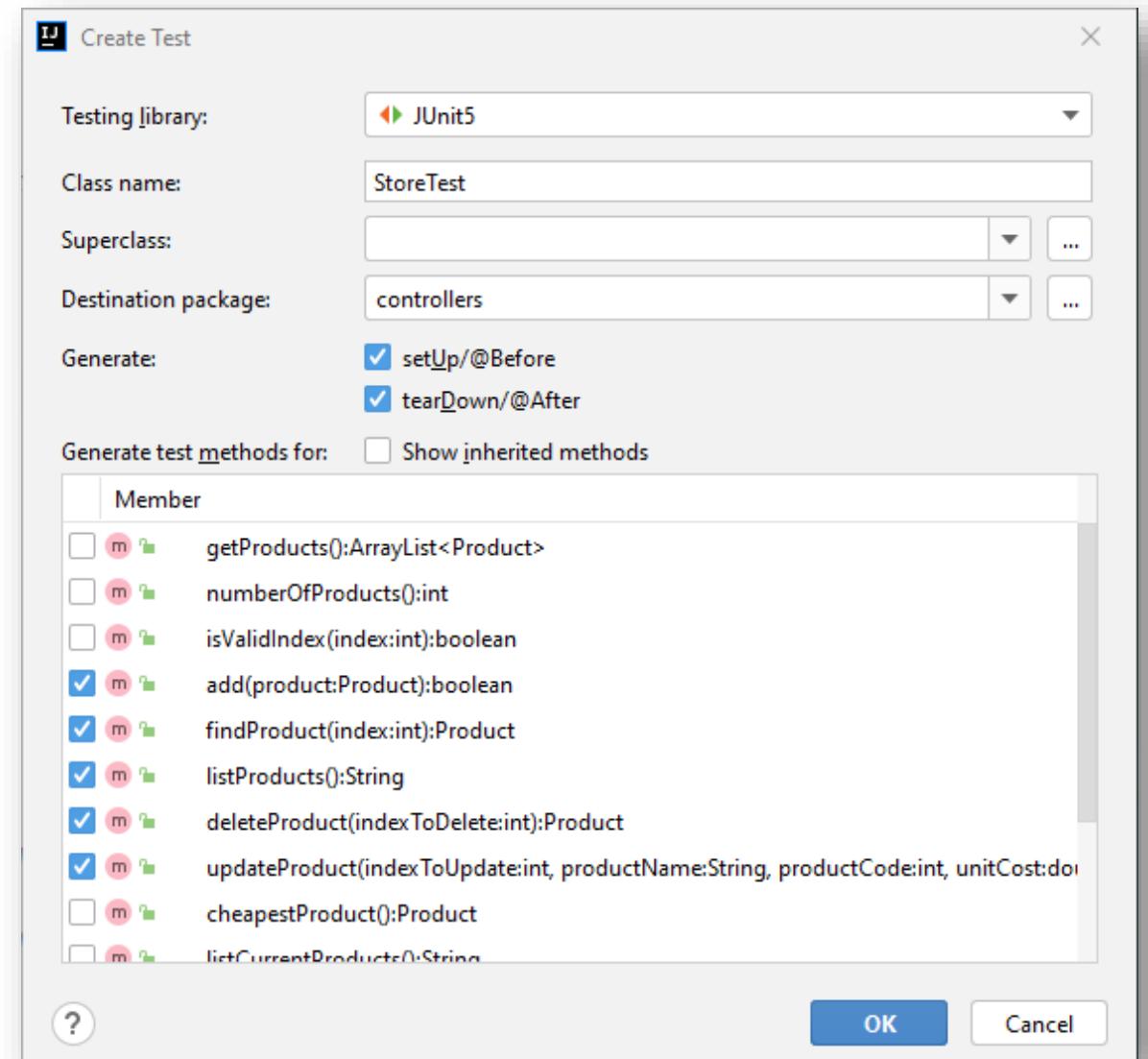
```
package com;
public class Store {
    public Store() { }
    public ArrayList<Product> getProducts() { }
    public int numberOfProducts() { }
    public boolean isValidIndex(int index) { }
    public boolean add(Product product) { }
    public Product findProduct(int index) { }
    public String listProducts() { }
    public Product deleteProduct(int index) { }
    public boolean updateProduct(int index, String name, int quantity, double price, boolean isOnSale) { }
    public Product cheapestProduct() { }
    public String listCurrentProducts() { }
    public double averageProductPrice() { }
    public String listProductsAboveAPrice(double price) { }
    public void load() { }
    public void save() { }
}
```

We will look at writing tests for a few of these methods.

Open Store.java and call “Create Test”

Call the test class,
StoreTest

Generate the default
setUp() and
tearDown() methods
and also generate
test methods for
selected member
methods.



The screenshot shows an IDE interface with the following details:

- Project Bar:** Shows the project name "ShopV6.0" and its path "D:\Siobhan\2021-2022\p".
- Toolbars:** Standard Java development toolbars.
- Left Sidebar:** "Project" view showing the project structure:
 - ShopV6.0
 - .idea
 - lib
 - xstream-1.4.17.jar
 - out
 - src
 - controllers
 - Store
 - main
 - Driver
 - models
 - Product
 - utils
 - ScannerInput
 - Utilities
 - test
 - controllers
 - StoreTest
 - models
 - ProductTest
 - utils
 - products.xml
 - ShopV6.0.iml
 - External Libraries
 - Scratches and Consoles
- Right Panel:** Code editor showing the generated `StoreTest.java` file. The code defines a test class `StoreTest` with various test methods and setup/teardown logic.

```
package controllers;

import org.junit.jupiter.api.AfterEach;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;

class StoreTest {

    @BeforeEach
    void setUp() {
    }

    @AfterEach
    void tearDown() {
    }

    @Test
    void add() {
    }

    @Test
    void findProduct() {
    }

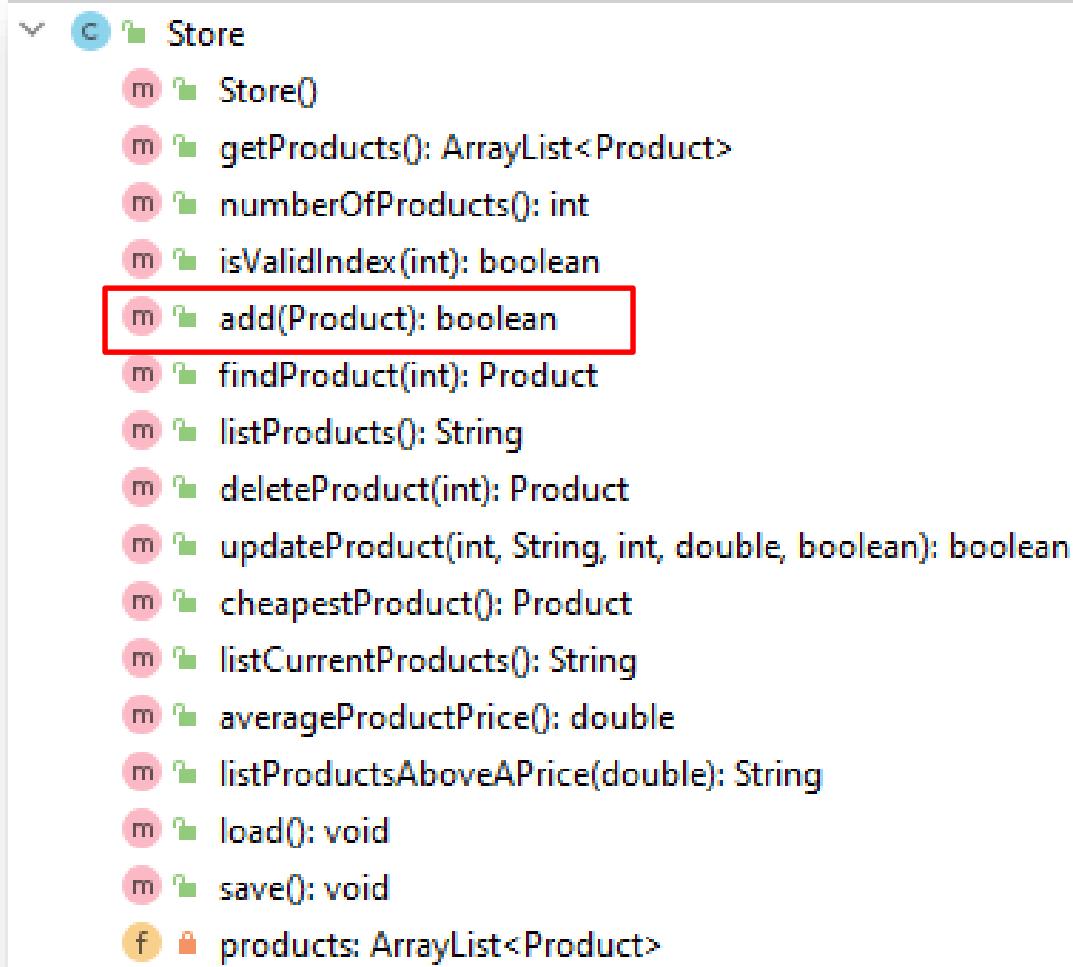
    @Test
    void listProducts() {
    }

    @Test
    void deleteProduct() {
    }

    @Test
    void updateProduct() {
    }
}
```

Generated StoreTest.java

Store.java – testing add(Product)



```
public boolean add (Product product){  
    return products.add (product);  
}
```

```

class StoreTest {

private Product productBelow, productExact, productAbove, productZero;
private Store storeWithProducts = new Store();
private Store storeEmpty = new Store();

@BeforeEach
void setUp() {
    //name, 19 chars, code 999, unitCost 1, inCurrentProductLine true.
    productBelow = new Product("Television 42Inches", 999, 1, true);
    //name, 20 chars, code 1000, unitCost 999, inCurrentProductLine true.
    productExact = new Product("Television 50 Inches", 1000, 999, true);
    //name, 21 chars, code 10000, unitCost 1000, inCurrentProductLine false.
    productAbove = new Product("Television 60 Inches.", 10000, 1000, false);
    //name, 0 chars, code 9999, unitCost 0, inCurrentProductLine false.
    productZero = new Product("", 9999, 0, false);

    storeWithProducts.add(productBelow);
    storeWithProducts.add(productExact);
    storeWithProducts.add(productAbove);
}

@AfterEach
void tearDown() {
    productBelow = productExact = productAbove = productZero = null;
    storeEmpty = storeWithProducts = null;
}
}

```

testing add(Product)

@Test

```

void addingToArrayListThatHasProductsIsSuccessful() {
    assertEquals(3, storeWithProducts.numberOfProducts());
    assertTrue(storeWithProducts.add(productZero));
    assertEquals(4, storeWithProducts.numberOfProducts());
    assertEquals(productZero, storeWithProducts.findProduct(3));
}

```

@Test

```

void addingToArrayListThatHasNoProductsIsSuccessful() {
    assertEquals(0, storeEmpty.numberOfProducts());
    assertTrue(storeEmpty.add(productZero));
    assertEquals(1, storeEmpty.numberOfProducts());
    assertEquals(productZero, storeEmpty.findProduct(0));
}

```

Store.java – testing deleteProduct(Product)

```
Store
  m Store()
  m getProducts(): ArrayList<Product>
  m int numberOfProducts()
  m boolean isValidIndex(int)
  m boolean add(Product)
  m Product findProduct(int)
  m String listProducts()
  m Product deleteProduct(int)
  m boolean updateProduct(int, String, int, double, boolean)
  m Product cheapestProduct()
  m String listCurrentProducts()
  m double averageProductPrice()
  m String listProductsAboveAPrice(double)
  m void load()
  m void save()
f ArrayList<Product> products
```

```
public Product deleteProduct(int indexToDelete) {
    if (isValidIndex(indexToDelete)) {
        return products.remove(indexToDelete);
    }
    return null;
}
```

```
class StoreTest {  
  
    private Product productBelow, productExact, productAbove, productZero;  
    private Store storeWithProducts = new Store();  
    private Store storeEmpty = new Store();  
  
    @BeforeEach  
    void setUp() {  
        //name, 19 chars, code 999, unitCost 1, inCurrentProductLine true.  
        productBelow = new Product("Television 42Inches", 999, 1, true);  
        //name, 20 chars, code 1000, unitCost 999, inCurrentProductLine true.  
        productExact = new Product("Television 50 Inches", 1000, 999, true);  
        //name, 21 chars, code 10000, unitCost 1000, inCurrentProductLine false.  
        productAbove = new Product("Television 60 Inches.", 10000, 1000, false);  
        //name, 0 chars, code 9999, unitCost 0, inCurrentProductLine false.  
        productZero = new Product("", 9999, 0, false);  
  
        storeWithProducts.add(productBelow);  
        storeWithProducts.add(productExact);  
        storeWithProducts.add(productAbove);  
    }  
  
    @AfterEach  
    void tearDown() {  
        productBelow = productExact = productAbove = productZero = null;  
        storeEmpty = storeWithProducts = null;  
    }  
}
```

testing deleteProduct(Product)

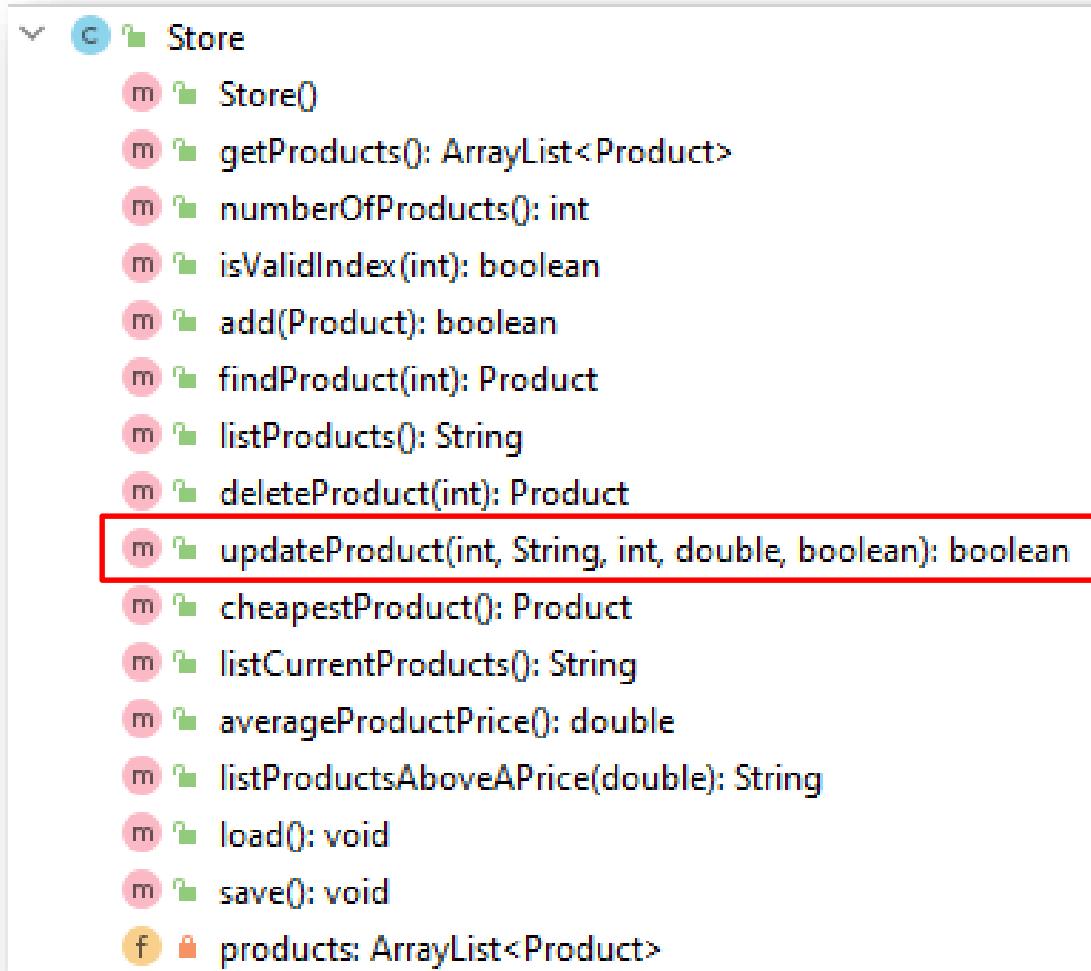
@Test

```
void deletingAProductThatDoesNotExistReturnsNull(){  
    assertNull(storeEmpty.deleteProduct(0));  
    assertNull(storeWithProducts.deleteProduct(-1));  
    assertNull(storeWithProducts.deleteProduct(3));  
}
```

@Test

```
void deletingAProductThatExistsDeletesAndReturnsDeletedObject()  
{  
    assertEquals(3, storeWithProducts.numberOfProducts());  
    assertEquals(productAbove, storeWithProducts.deleteProduct(2));  
    assertEquals(2, storeWithProducts.numberOfProducts());  
}
```

Store.java – testing updateProduct



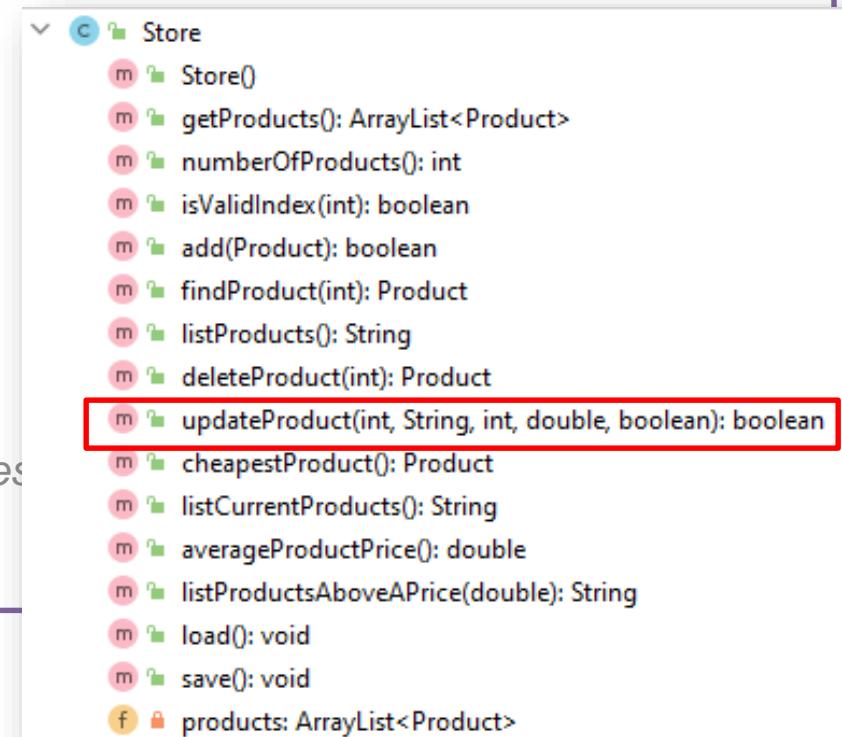
```
C  Store
m  Store()
m  getProducts(): ArrayList<Product>
m  numberOfRowsInSection(): int
m  isValidIndex(int): boolean
m  add(Product): boolean
m  findProduct(int): Product
m  listProducts(): String
m  deleteProduct(int): Product
m  updateProduct(int, String, int, double, boolean): boolean
m  cheapestProduct(): Product
m  listCurrentProducts(): String
m  averageProductPrice(): double
m  listProductsAboveAPrice(double): String
m  load(): void
m  save(): void
f  products: ArrayList<Product>
```

Store.java – testing updateProduct

```
public boolean updateProduct(int indexToUpdate, String productName, int productCode, double unitCost, boolean inCurrentProductLine) {
    //find the product object by the index number
    Product foundProduct = findProduct(indexToUpdate);

    //if the product exists, use the details passed in the updateDetails parameter to
    //update the found product in the ArrayList.
    if (foundProduct != null) {
        foundProduct.setProductName(productName);
        foundProduct.setProductCode(productCode);
        foundProduct.setUnitCost(unitCost);
        foundProduct.setInCurrentProductLine(inCurrentProductLine);
        return true;
    }

    //if the product was not found, return false, indicating that the update was not successful
    return false;
}
```



```

class StoreTest {

private Product productBelow, productExact, productAbove, productZero;
private Store storeWithProducts = new Store();
private Store storeEmpty = new Store();

@BeforeEach
void setUp() {
    //name, 19 chars, code 999, unitCost 1, inCurrentProductLine true.
    productBelow = new Product("Television 42Inches", 999, 1, true);
    //name, 20 chars, code 1000, unitCost 999, inCurrentProductLine true.
    productExact = new Product("Television 50 Inches", 1000, 999, true);
    //name, 21 chars, code 10000, unitCost 1000, inCurrentProductLine false.
    productAbove = new Product("Television 60 Inches.", 10000, 1000, false);
    //name, 0 chars, code 9999, unitCost 0, inCurrentProductLine false.
    productZero = new Product("", 9999, 0, false);

    storeWithProducts.add(productBelow);
    storeWithProducts.add(productExact);
    storeWithProducts.add(productAbove);
}

@AfterEach
void tearDown() {
    productBelow = productExact = productAbove = productZero = null;
    storeEmpty = storeWithProducts = null;
}
}

```

testing updateProduct

@Test

```

void updatingANoteThatExistsReturnsTrueAndUpdates(){
    //check product index 2 exists and check the contents
    assertEquals(productAbove, storeWithProducts.findProduct(2));
    assertEquals("Television 60 Inches", storeWithProducts.findProduct(2).getProductName());
    assertEquals(-1, storeWithProducts.findProduct(2).getProductCode());
    assertEquals(1000, storeWithProducts.findProduct(2).getUnitCost());
    assertFalse(storeWithProducts.findProduct(2).isInCurrentProductLine());

    //update product 2 with new information and ensure contents updated successfully
    assertTrue(storeWithProducts.updateProduct(2, "Updating Product", 2000, 19.99, true));
    assertEquals("Updating Product", storeWithProducts.findProduct(2).getProductName());
    assertEquals(2000, storeWithProducts.findProduct(2).getProductCode());
    assertEquals(19.99, storeWithProducts.findProduct(2).getUnitCost());
    assertTrue(storeWithProducts.findProduct(2).isInCurrentProductLine());
}

```

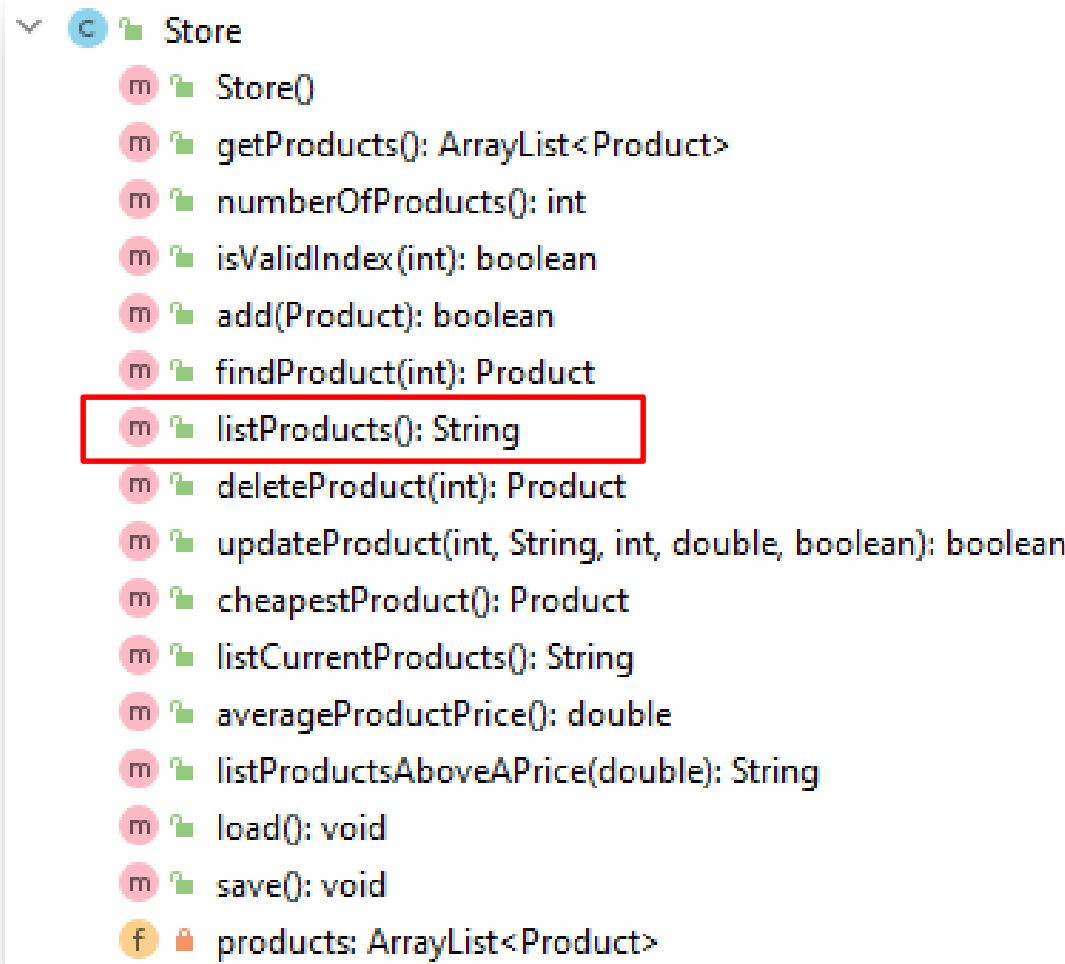
@Test

```

void updatingAProductThatDoesNotExistReturnsFalse(){
    assertFalse(storeWithProducts.updateProduct(3, "Updating Product", 2, 19.99, false));
    assertFalse(storeWithProducts.updateProduct(-1, "Updating Product", 1002, 14.49, true));
    assertFalse(storeEmpty.updateProduct(0, "Updating Product", 1003, 199.99, false));
}

```

Store.java – testing listProduct()



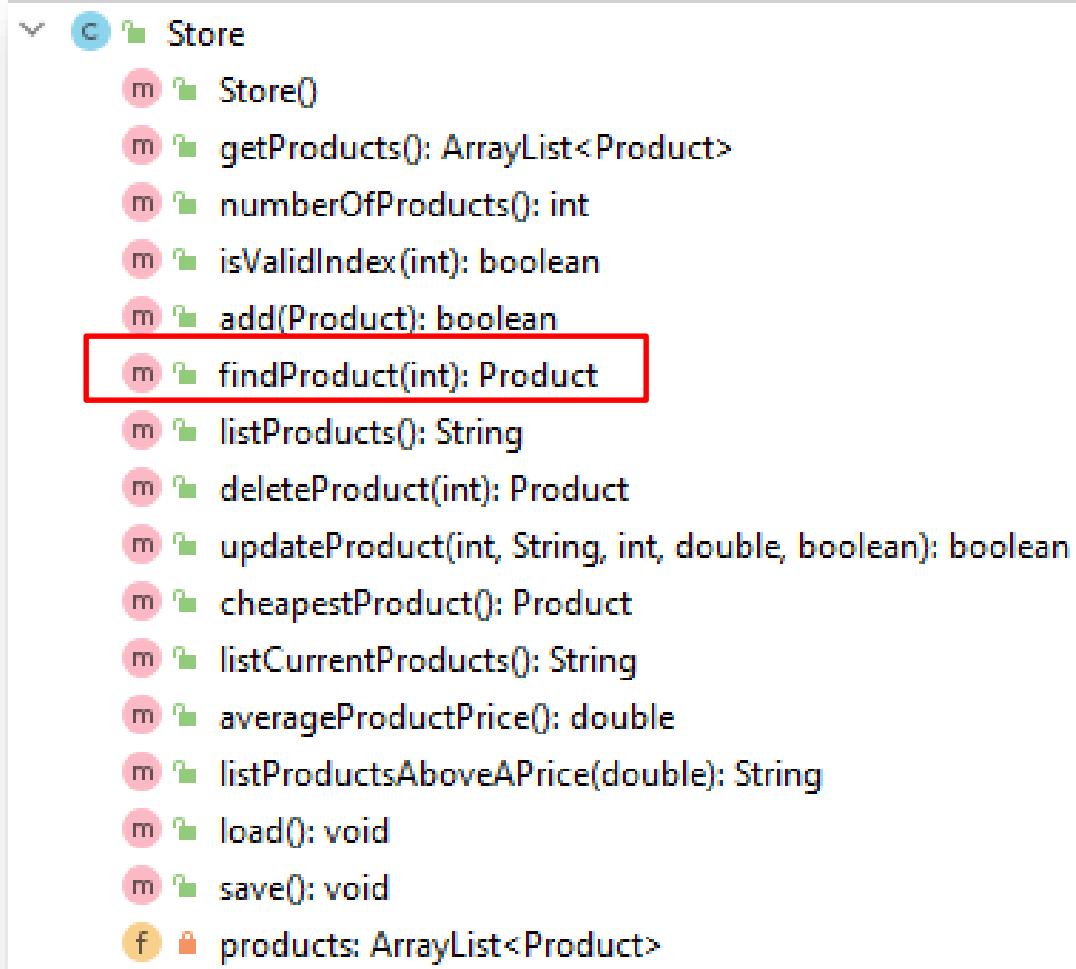
```
public String listProducts() {
    if (products.isEmpty()) {
        return "No products in the store";
    } else {
        String listOfProducts = "";
        for (int i = 0; i < products.size(); i++) {
            listOfProducts += i + ": " + products.get(i) + "\n";
        }
        return listOfProducts;
    }
}
```

```
class StoreTest {  
  
    private Product productBelow, productExact, productAbove, productZero;  
    private Store storeWithProducts = new Store();  
    private Store storeEmpty = new Store();  
  
    @BeforeEach  
    void setUp() {  
        //name, 19 chars, code 999, unitCost 1, inCurrentProductLine true.  
        productBelow = new Product("Television 42Inches", 999, 1, true);  
        //name, 20 chars, code 1000, unitCost 999, inCurrentProductLine true.  
        productExact = new Product("Television 50 Inches", 1000, 999, true);  
        //name, 21 chars, code 10000, unitCost 1000, inCurrentProductLine false.  
        productAbove = new Product("Television 60 Inches.", 10000, 1000, false);  
        //name, 0 chars, code 9999, unitCost 0, inCurrentProductLine false.  
        productZero = new Product("", 9999, 0, false);  
  
        storeWithProducts.add(productBelow);  
        storeWithProducts.add(productExact);  
        storeWithProducts.add(productAbove);  
    }  
  
    @AfterEach  
    void tearDown() {  
        productBelow = productExact = productAbove = productZero = null;  
        storeEmpty = storeWithProducts = null;  
    }  
}
```

testing listProduct()

```
@Test  
void listProductsReturnsNoProductsStoredWhenArrayListIsEmpty() {  
    assertEquals(0, storeEmpty.numberOfProducts());  
    assertTrue(storeEmpty.listProducts().toLowerCase().contains("no products"))  
}  
  
@Test  
void listProductsReturnsProductsWhenArrayListHasProductsStored() {  
    assertEquals(3, storeWithProducts.numberOfProducts());  
    String productsString = storeWithProducts.listProducts();  
    assertTrue(productsString.contains("Television 42Inches"));  
    assertTrue(productsString.contains("Television 50 Inches"));  
    assertTrue(productsString.contains("Television 60 Inches"))  
}
```

Store.java – testing findProduct(int)



```
public Product findProduct(int index) {  
    if (isValidIndex(index)) {  
        return products.get(index);  
    }  
    return null;  
}
```

```
class StoreTest {  
  
    private Product productBelow, productExact, productAbove, productZero;  
    private Store storeWithProducts = new Store();  
    private Store storeEmpty = new Store();  
  
    @BeforeEach  
    void setUp() {  
        //name, 19 chars, code 999, unitCost 1, inCurrentProductLine true.  
        productBelow = new Product("Television 42Inches", 999, 1, true);  
        //name, 20 chars, code 1000, unitCost 999, inCurrentProductLine true.  
        productExact = new Product("Television 50 Inches", 1000, 999, true);  
        //name, 21 chars, code 10000, unitCost 1000, inCurrentProductLine false.  
        productAbove = new Product("Television 60 Inches.", 10000, 1000, false);  
        //name, 0 chars, code 9999, unitCost 0, inCurrentProductLine false.  
        productZero = new Product("", 9999, 0, false);  
  
        storeWithProducts.add(productBelow);  
        storeWithProducts.add(productExact);  
        storeWithProducts.add(productAbove);  
    }  
  
    @AfterEach  
    void tearDown() {  
        productBelow = productExact = productAbove = productZero = null;  
        storeEmpty = storeWithProducts = null;  
    }  
}
```

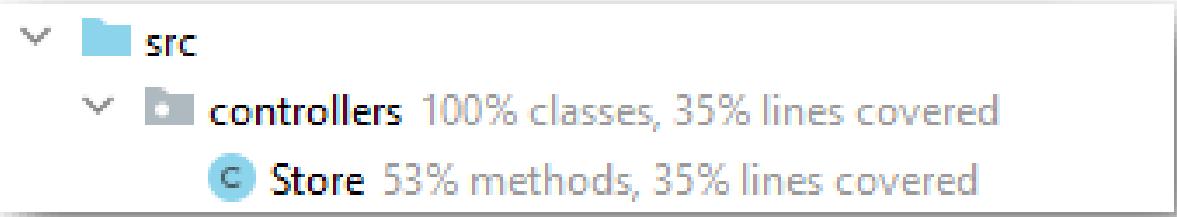
testing findProduct(int)

```
@Test  
void findProductReturnsProductWhenIndexIsValid() {  
    assertEquals(3, storeWithProducts.numberOfProducts());  
    assertEquals(productBelow, storeWithProducts.findProduct(0));  
    assertEquals(productAbove, storeWithProducts.findProduct(2));  
}  
  
@Test  
void findProductReturnsNullWhenIndexIsInvalid() {  
    assertEquals(0, storeEmpty.numberOfProducts());  
    assertNull(storeEmpty.findProduct(0));  
    assertEquals(3, storeWithProducts.numberOfProducts());  
    assertNull(storeWithProducts.findProduct(-1));  
    assertNull(storeWithProducts.findProduct(3));  
}
```

Run: StoreTest

The screenshot shows a test results window with the following details:

- Test Results:** A tree view showing the test structure:
 - StoreTest**: Contains:
 - FindAndSearch**: Contains:
 - findProductReturnsProductWhenIndexIsValid()
 - findProductReturnsNullWhenIndexIsInvalid()
 - ArrayListCRUD**: Contains:
 - deletingAProductThatExistsDeletesAndReturnsDeletedObject()
 - deletingAProductThatDoesNotExistReturnsNull()
 - updatingANoteThatExistsReturnsTrueAndUpdates()
 - addingToAnArrayListThatHasProductsIsSuccessful()
 - listProductsReturnsNoProductsStoredWhenArrayListIsEmpty()
 - updatingAProductThatDoesNotExistReturnsFalse()
 - addingToAnArrayListThatHasNoProductsIsSuccessful()
 - listProductsReturnsProductsWhenArrayListHasProductsStored()
- Time:** 17 ms
- Path:** D:\Siobhan\dev\Java\bin\java.exe ...
- Status:** Tests passed: 10 of 10 tests – 17 ms
- Message:** Process finished with exit code 0



The screenshot shows a Java code editor with a file named `Store.java` open. The code defines a class `Store` with methods to get products and check if an index is valid. A coverage analysis window is overlaid on the right side of the editor.

```
20
21  public class Store {
22      private ArrayList<Product> products;
23
24
25      public Store(){
26          products = new ArrayList<Product>();
27      }
28
29      public ArrayList<Product> getProducts() {
30          return products;
31      }
32
33      /**
34      * This method returns the number of product objects stored in the ArrayList.
35      *
36      * @return An int value representing the number of product objects in the ArrayList.
37      */
38      public int numberOfProducts() { return products.size(); }
39
40
41      /**
42      * This method takes in a number and checks if it is a valid index in the products ArrayList.
43      *
44      * @param index A number representing a potential index in the ArrayList.
45      * @return True if the index number passed is a valid index in the ArrayList, false otherwise.
46      */
47      public boolean isValidIndex(int index) {
48          return (index >= 0) && (index < products.size());
49      }
50
51  }
```

Coverage analysis results:

- SRC**: controllers 100% classes, 35% lines covered
- Store**: 53% methods, 35% lines covered

Topic List

- Product and ProductTest.java
- JUnit Testing of Store.java
- Testing Driver.java

Driver.java (and ScannerInput.java)

- JUnit is not used to test the class that takes input from the console.
- Why do you think this is?

Any
Questions?

