ShopV2.2

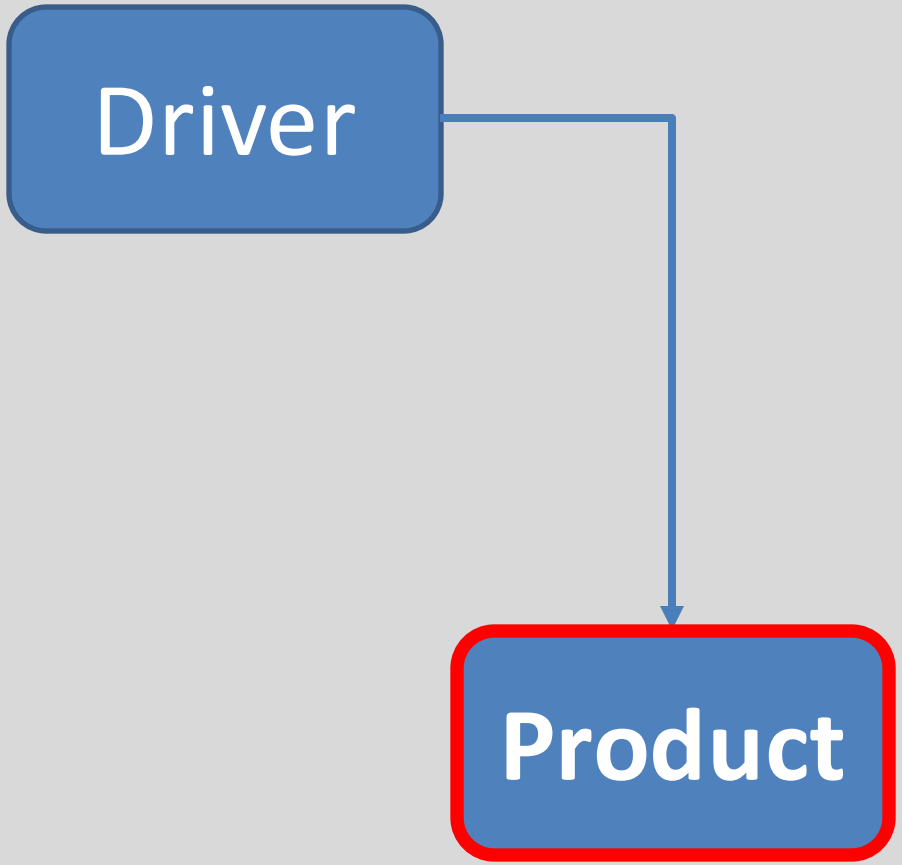
Improving Product(with Validation)

Produced by: Dr. Siobhán Drohan,
Ms. Mairead Meagher,
Ms. Siobhán Roche.

Ollscoil Teicneolaíochta an Oirdheiscirt
South East Technological University

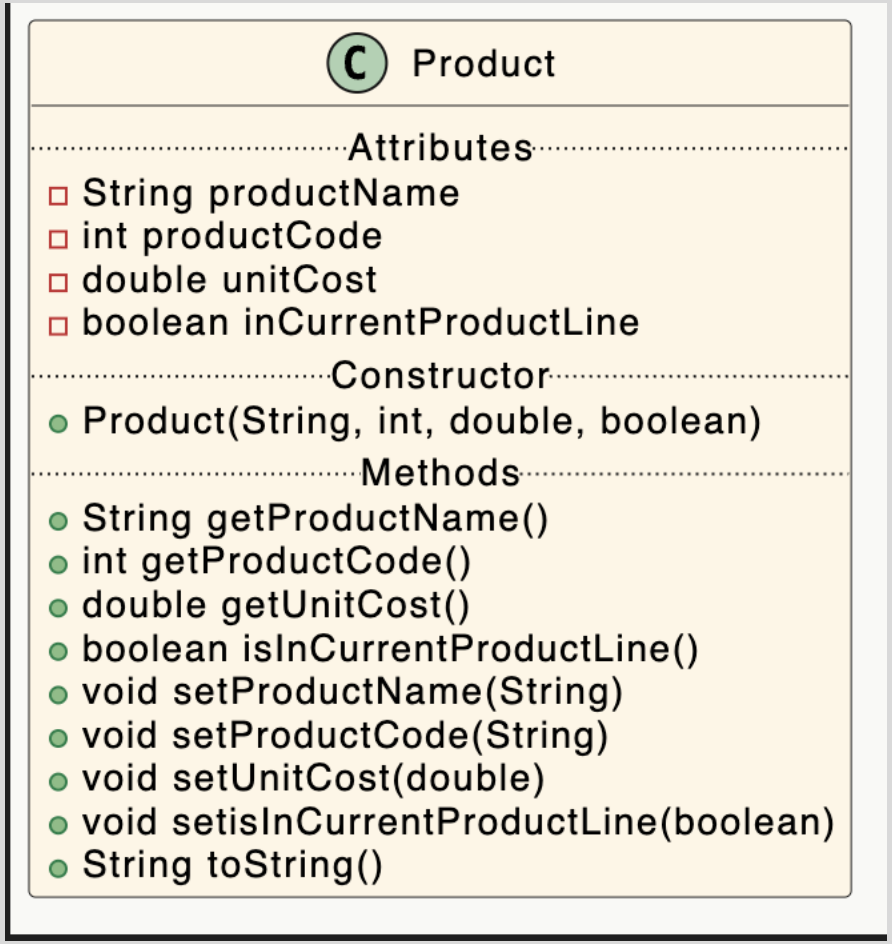
Department of Computing and Mathematics
http://www.setu.ie/

Recap: Shop V1.0 - **Product**



- The **Product** class stores **details** about a product:
  - name
  - code
  - unit cost
  - in the current product line or not?

The **Product** Class…

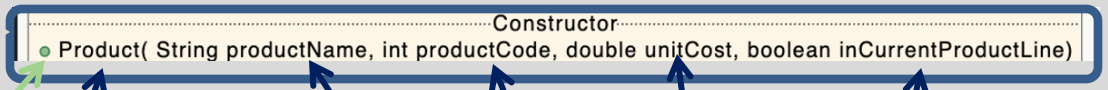


A **Product** Class… **constructor**

Constructor
i.e. for building objects.

The green circle means it is **public**.

Constructors have same name as the class



Four **parameters**;
one for each field.

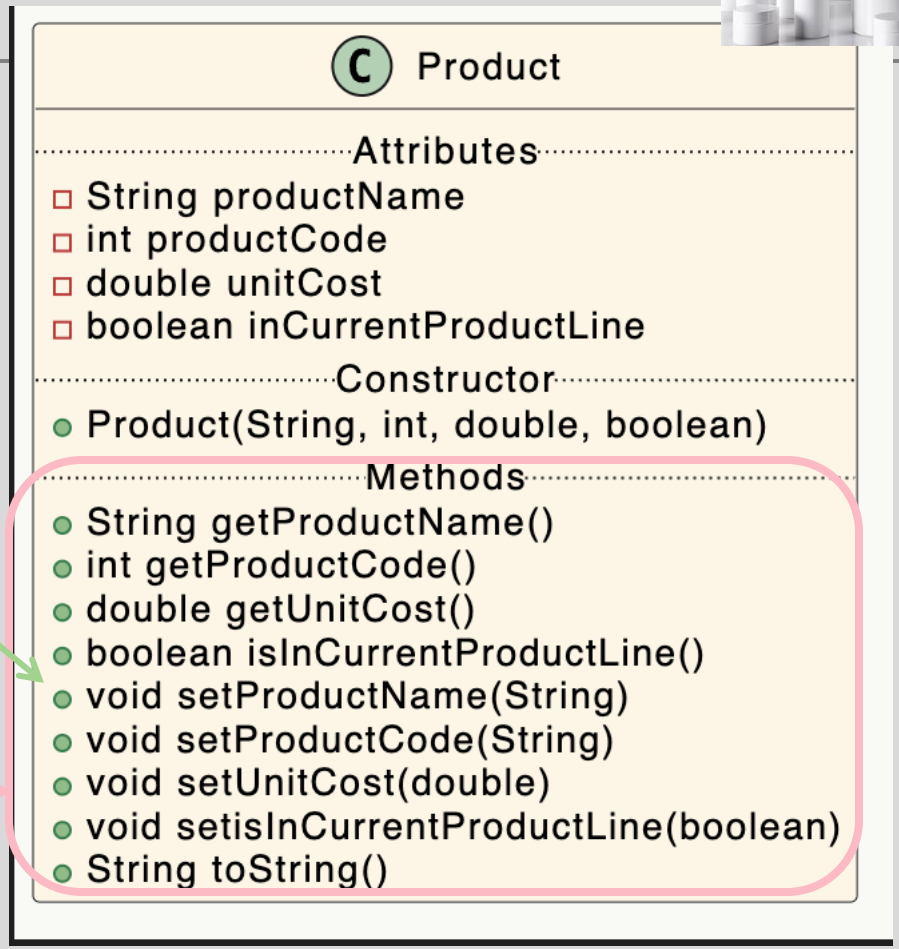
A **Product** Class… **fields** and **constructor**

```
public class Product {

    private String productName;
    private int productCode;
    private double unitCost;
    private boolean inCurrentProductLine;

    public Product (String productName, int productCode,
                    double unitCost, boolean inCurrentProductLine){

        this.productName = productName;
        this.productCode = productCode;
        this.unitCost = unitCost;
        this.inCurrentProductLine = inCurrentProductLine;
    }
```

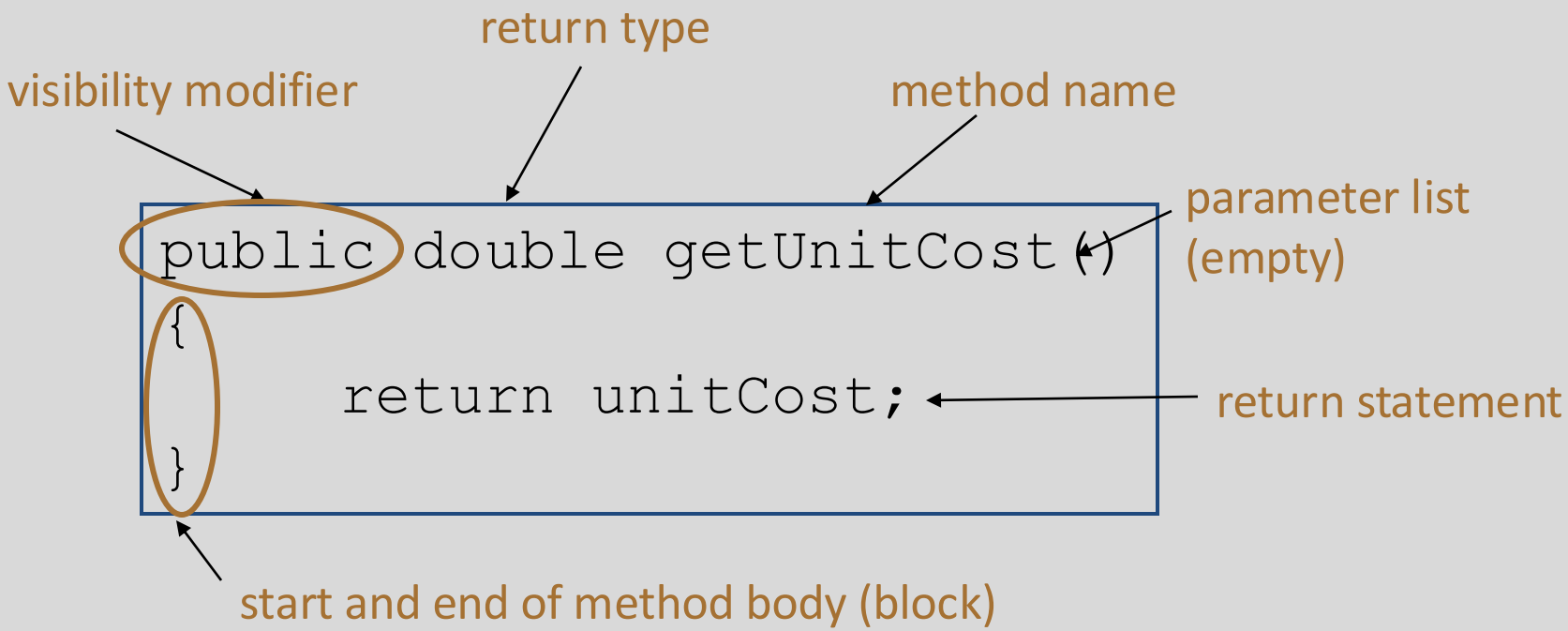
A Product Class… **methods**

The green circle means it is **public**.

Methods
i.e. the **behaviours** of the class



Getters

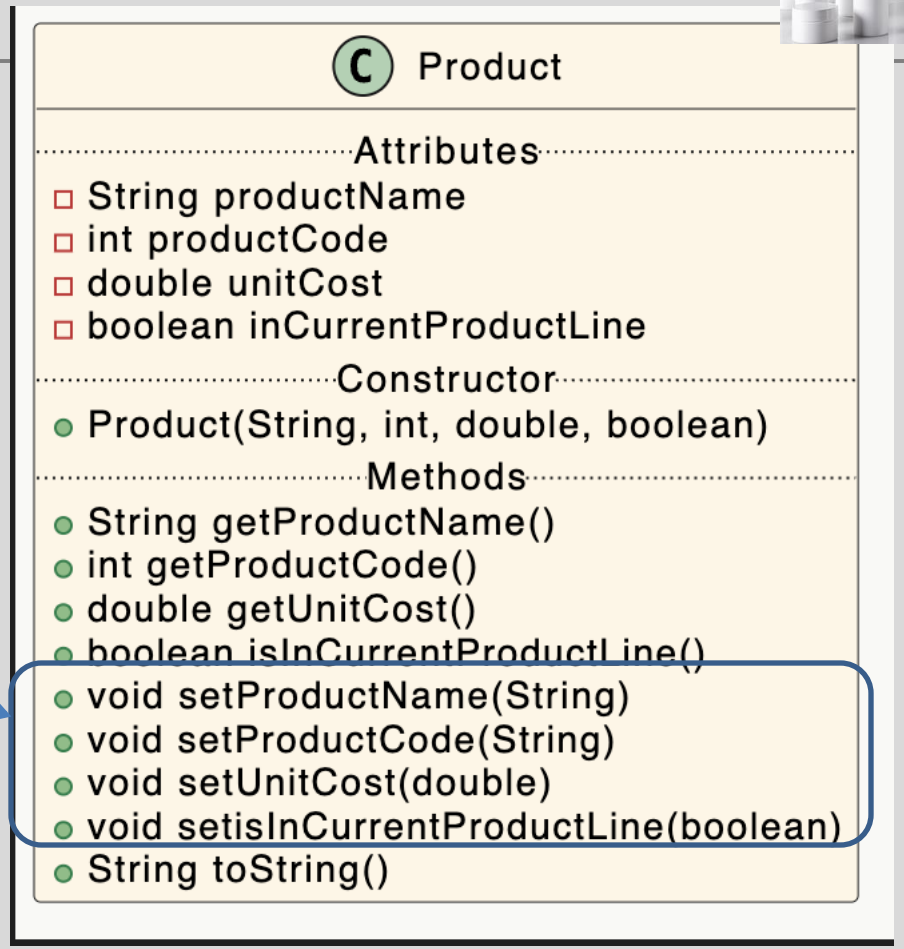

return type
visibility modifier
method name
parameter list
(empty)
public double getUnitCost()
{
return unitCost;
}
return statement
start and end of method body (block)

A Product Class…**get**ters

```
public String getProductName(){
    return productName;
}

public double getUnitCost(){
    return unitCost;
}

public int getProductCode() {
    return productCode;
}

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

A Product Class…**set**ters

setters



Setters

return type

visibility modifier

method name

parameter

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

field being mutated

assignment statement

Value passed as a parameter

A Product Class…setters

```
public void setProductCode(int productCode) {
    this.productCode = productCode;
}

public void setProductName(String productName) {
    this.productName = productName;
}

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

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

Getters/Setters

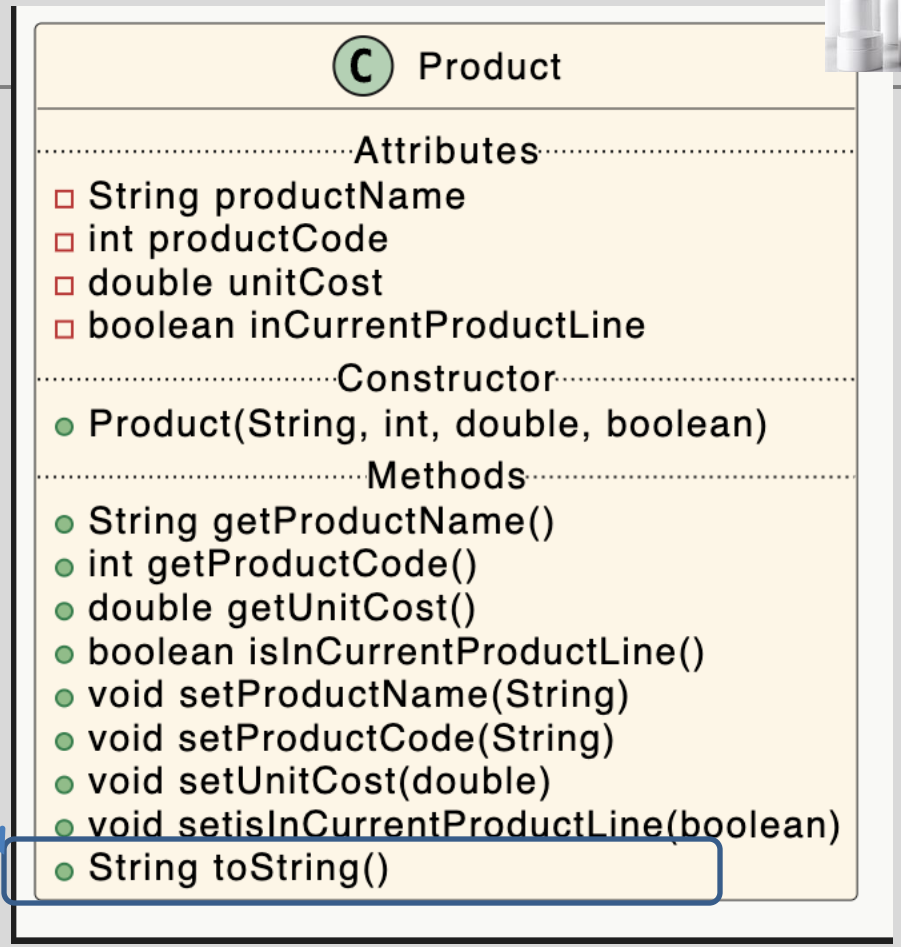
For each instance field in a class, you are normally asked to write:

- A **get**ter
  - Return statement
- A **set**ter
  - Assignment statement

A Product Class…**toString**

toString():

Builds and returns a String containing a user-friendly representation of the object state.



A Product Class…

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

Sample Console Output if we printed a Product Object:

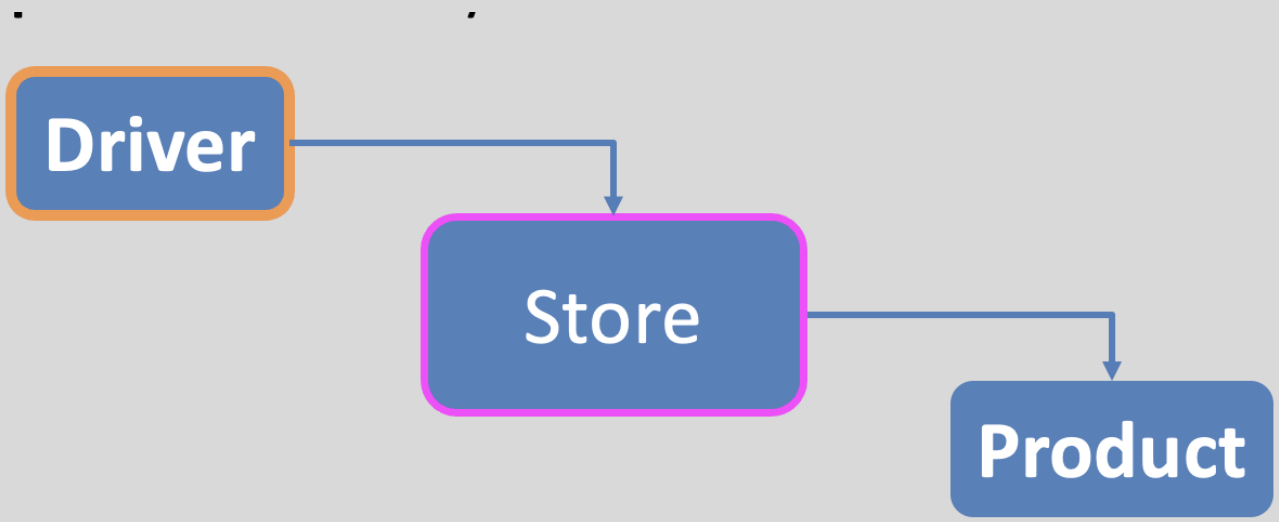
Product description: 24 Inch TV, product code: 23432, unit cost: 399.99, currently in product line: true

Shop V2.1 – Sample I/O

Entering a product:

```
Enter the Product Name:  LG OLED TV 55 inch Model XXY
Enter the Product Code:  2500
Enter the Unit Cost:  999.00
Is this product in your current line (y/n): y
```

Shop V2.1



1. Invites the user to enter the details of a product

2. Prints the details about that product to the console

Limitations:

1. Currently no benefit from all the infrastructure
2. No extra checks on the data.
3. We could do more checking on the data coming in (from keyboard) in case there are obvious errors.

No Validation

- The purpose of this app is to read in a product, and later perform calculations on it, adding it to collections, updating it etc.

- You could, for either the add or the update, enter preposterous values for any of the fields and at the moment we simply accept them

```
Enter the Product Name:  LG OLED TV 55 inch Model XXY
Enter the Product Code:  2500
Enter the Unit Cost:  999.00
Is this product in your current line (y/n): y
```

No Validation

- Clearly we need to put some restrictions in place for each field e.g. :
  - min and max permitted values
  - default value if the min and max are breached
  - Strings have a max length

```
Enter the Product Name:  LG OLED TV 55 inches
Enter the Product Code:  3456
Enter the Unit Cost:  950.0
Is this product in your current line (y/n): y
```

Validation Rules for fields in the Product class

Field Name	Field type	Rules for valid values	Default Value	Note
productName	String	Length of string <= 20	In constructor, if the productName is longer than 20 characters, the string should be truncated to 20	This is an outlier – for other types the default is not at all related to the invalid input.
productCode	int	Valid values are between 1000 and 9999 (inclusive)	9999	This is a nice way of ensuring 4 digits
unitCost	double	Non-negative (>=0)	0	
inCurrentProductionLine	boolean	No validation required	false	You should never need validation on Booleans.

Implementing Validation Rules

Default Values

Validation Rules – implementing default values

Field Name	Default Value
productName	In constructor, if the productName is longer than 20 characters, the string should be truncated to 20
productCode	9999
unitCost	0
inCurrentProductionLine	false

```
public class Product {
    private String productName;
    private int productCode;
    private double unitCost;
    private boolean inCurrentProductLine;
```

becomes

```
public class Product {
    private String productName = "";
    private int productCode = 9999;
    private double unitCost = 0;
    private boolean inCurrentProductLine = false;
```

Implementing Validation Rules

productCode

productCode – mutator changes

Field	Min Value	Max Value
productCode	Greater than or equal to 1000	Less than or equal to 9999

```
public void setProductCode(int productCode) {
        this.productCode = productCode;
}
```

becomes

```
public void setProductCode(int productCode) {
    if ((productCode >= 1000) && productCode <= 9999 ) {
        this.productCode = productCode;
    }
}
```

productCode – constructor changes

Field	Min Value	Max Value	Default value
productCode	Greater than or equal to 1000	Less than or equal to 9999	9999

```
public Product(String productName, int productCode, double unitCost, boolean inCurrentProductLine) {
    this.productName = productName;
    this.productCode = productCode;
    this.unitCost = unitCost;
    this.inCurrentProductLine = inCurrentProductLine;
}
```

becomes

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

Implementing Validation Rules

unitCost

unitCost – mutator changes

Field	Validation Rule
unitCost	Non-negative (>=0)

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

becomes

```
public void setUnitCost(double unitCost) {
    if (unitCost>= 0)
        this.unitCost = unitCost;
}
```

unitCost – constructor changes

Field	Validation Rule	Default
unitCost	Greater than or equal to 0	0

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

becomes

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

Implementing Validation Rules

productName

Implementation of validation with Strings

- Strings use a different rule for default values when the validation rule is based on length of string.
- Numbers, chars, etc. use a fixed default so we can use setters in the constructor (if the values are invalid, the default values remain)
- If the String value we are checking is invalid, then instead of staying at a fixed default value, we (usually) truncate the input String to the correct length

productName – mutator changes

Field	Validation Rule
productName	Length of string must be <= 20

```
public void setProductName(String productName
        this.productName = productName;
}
```

becomes

```
public void setProductName(String productName) {
    if (productName.length() <= 20) {
        this.productName = productName;
    }
}
```

productName – constructor changes

Field	Validation Rule	Default
productName	Length of string must be <= 20	If length of string is > 20, use first 20 chars of string

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

Setter not used here

becomes

```
public Product(String productName, int productCode, double unitCost, boolean inCurrentProductLine) {
    if (productName.length()<= 20)
        this.productName = productName;
    else
        this.productName = productName.substring(0,20);
    setProductCode(productCode);
    setUnitCost(unitCost);
    this.inCurrentProductLine = inCurrentProductLine;
}
```

inCurrentProductionLine

- There is no validation needed for inCurrentProductionLine
- This is generally the case for booleans.

Boundary Testing the Validation Rules

For BOTH add and update

What is Boundary Testing?

Boundary Testing is when you input test data that are:

Just above	Just at lower bound	in middle of valid range
Just below	Just at upper bound	

the boundary values in your Boolean expressions.

Inputting the following values for productCode would test the 'boundaries' of this if statement:

[999, 1000, 2000, 9999, 10000]

tests

```
public void setProductCode(int productCode) {
    if ((productCode >= 1000) && productCode <= 9999 ) {
        this.productCode = productCode;
    }
}
```

Example Boundary Test for productCode

Just-Below Upper Boundary Test:

- default values are correctly used when adding and updating.

Add a new Product with the value of

999

for productCode

The default value of 9999 should be stored

Example Boundary Test

Just-Inside Lower Boundary Test:

- values are accepted for both add and update

Add a new Product with the value of

1000

for productCode

The value of 1000 should be accepted and stored

Example Boundary Test

Inside range Boundary Test:

- values are accepted for both add and update

Add a new Product with the value of

2000

for productCode

The value of 2000 should be accepted and stored

Example Boundary Test

Just-Below Upper range Boundary Test:

- values are accepted for both add and update

Add a new Product with the value of

9999

for productCode

The value of 9999 should be accepted and stored

Example Boundary Test

Just-Above Upper range Boundary Test:

- values are not accepted for either add or update

Add a new / updated a currentProduct with the value of

10000

for productCode

For adding (**constructor)**, 9999 (default) should be stored,
For **update** (setter) the previous value should remain unchanged.

The rest of the fields

The checking of the other fields are left as an exercise.

Note that later, once you are familiar with the way to test these boundaries, we will use JUnit to help automate the running (and re-running) of these checks.

Questions?