

# More on ArrayLists

(based on Ch. 4, Objects First with Java - A Practical  
Introduction using BlueJ, © David J. Barnes, Michael Kölling)

---

Produced by:     Dr. Siobhán Drohan  
                      Ms. Mairéad Meagher  
                      Ms Siobhan Roche

# Topic list

---

- Grouping Objects
  - Developing a basic personal notebook project using Collections e.g. ArrayList
- Indexing within Collections
  - Retrieval and removal of objects
- Generic classes e.g. ArrayList
- Iteration
  - Using the for loop
  - Using the while loop
  - Using the for each loop
- **ShopV3.0 – use an ArrayList of Products instead of an array.**

# RECAP: Shop V2.0



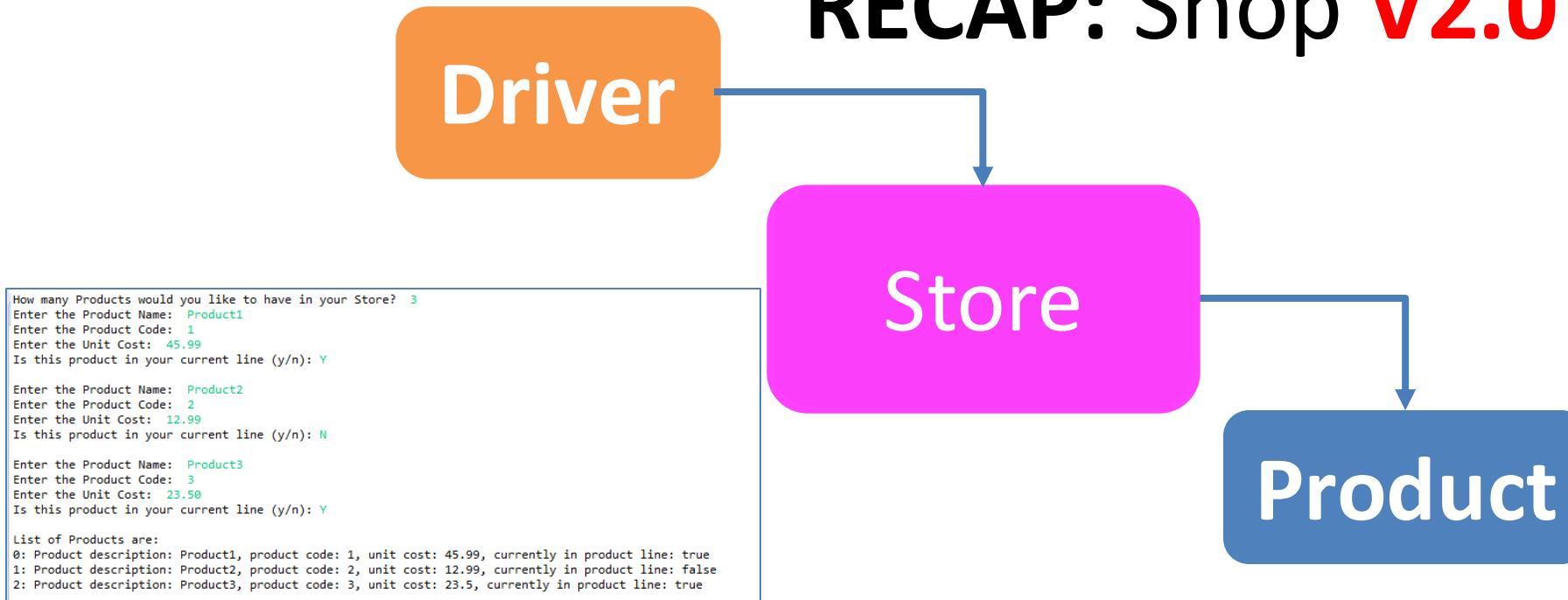
Product class stores details of a product's name, code, unit cost and whether it is in the current product line or not.

# RECAP: Shop V2.0



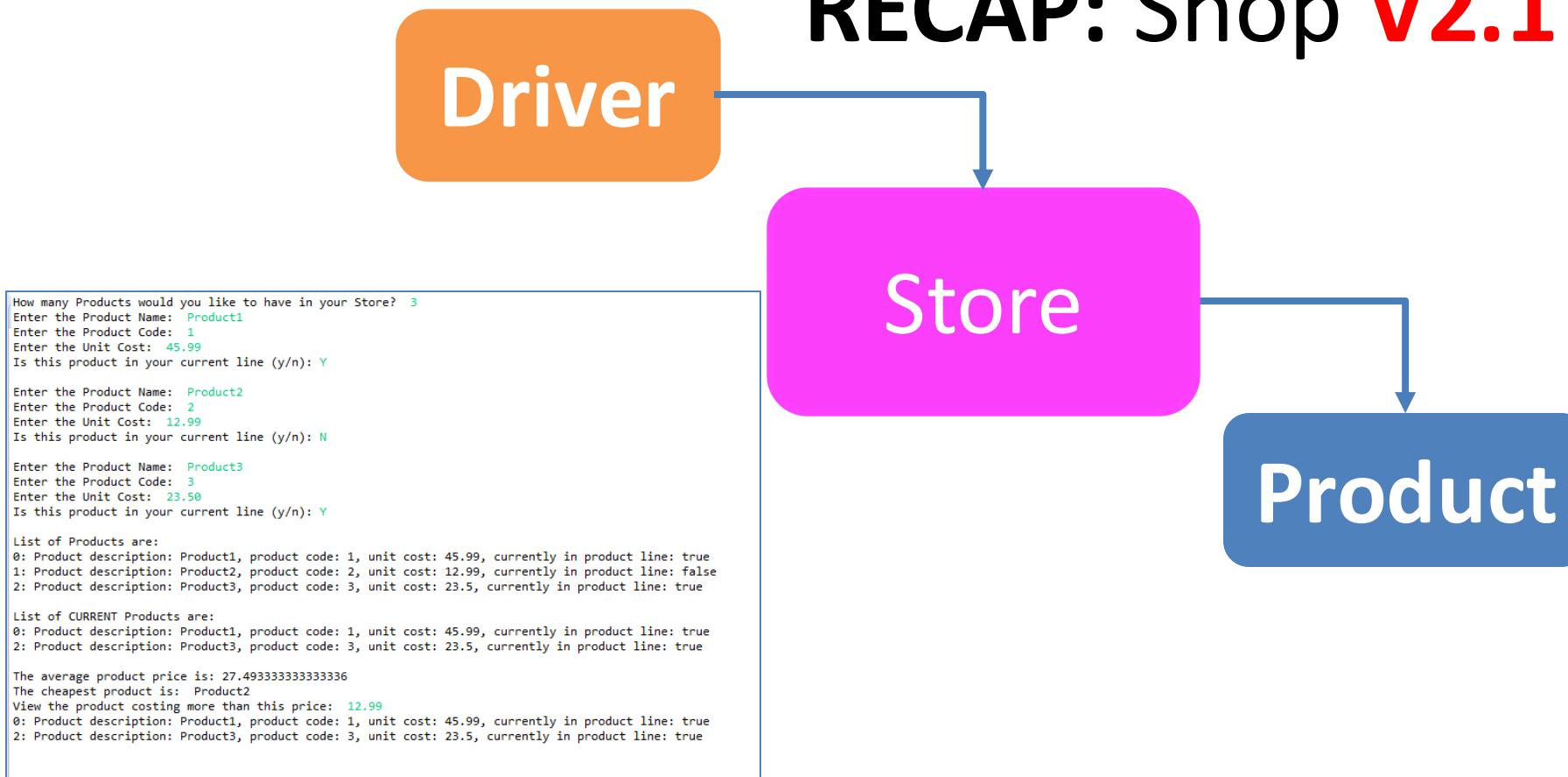
**Store** class maintains a collection of Products  
i.e. a **primitive array of Products**; `store.products[]`

# RECAP: Shop V2.0



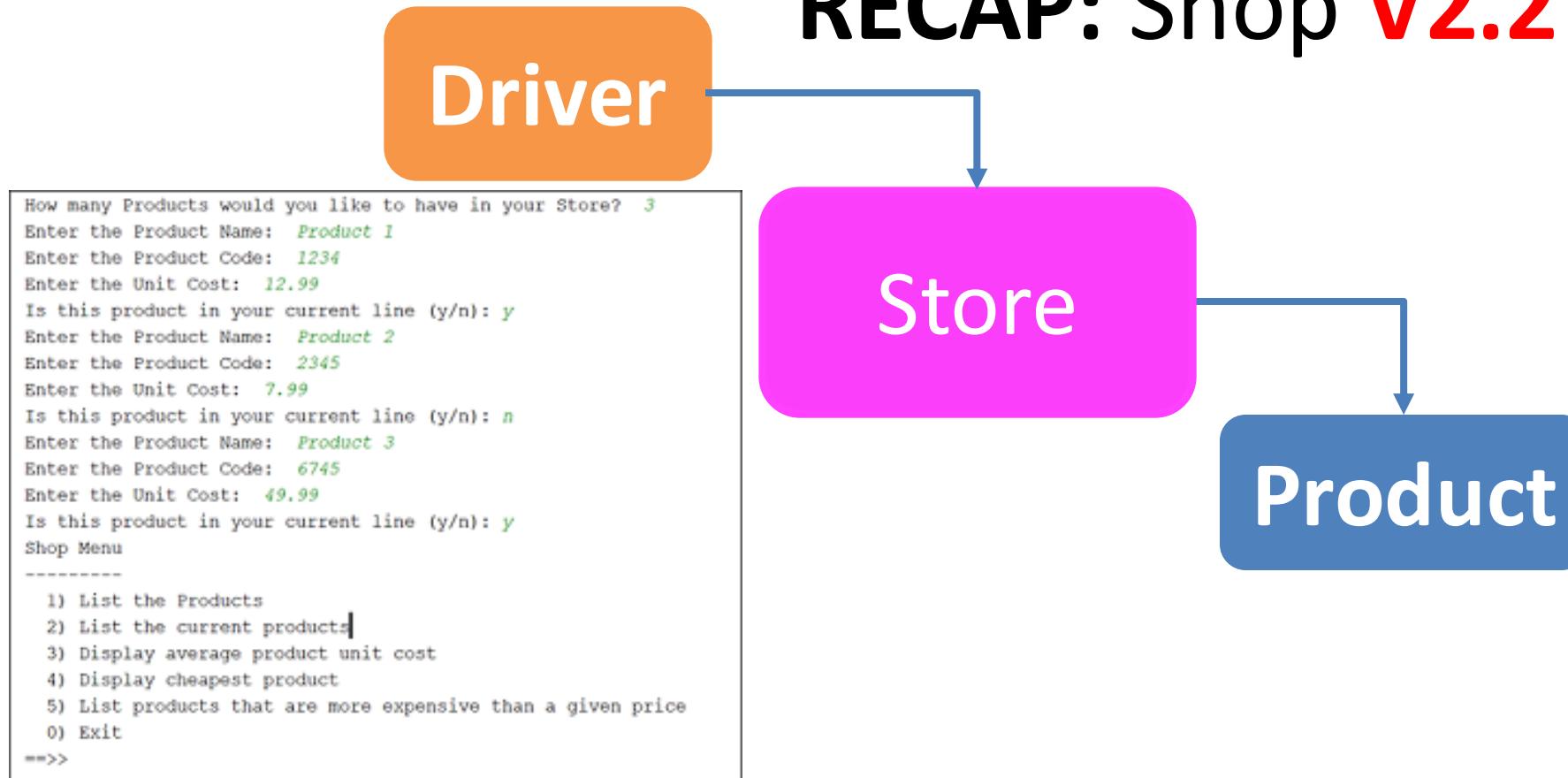
**Driver** allows the user to decide **how many product** details they want to store, then it stores the product details and prints them out.

# RECAP: Shop V2.1



**Driver** allows the user to decide **how many product** details they want to store, then it stores the product details and prints them out.  
**AND also provides methods operating on the array**

# RECAP: Shop V2.2



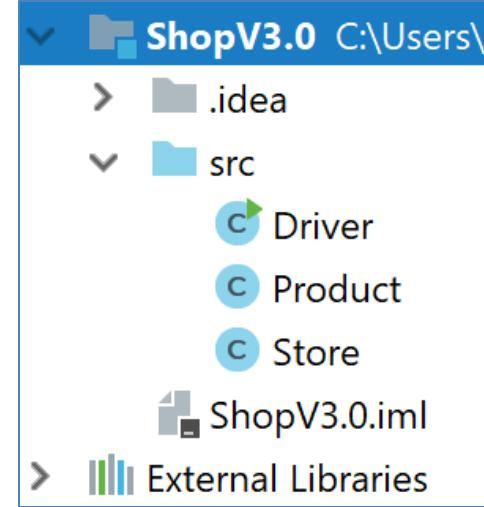
**Driver** allows the user to decide **how many product** details they want to store, then it stores the product details and prints them out.

**AND also provides methods operating on the array**

**AND provides a console menu system.**



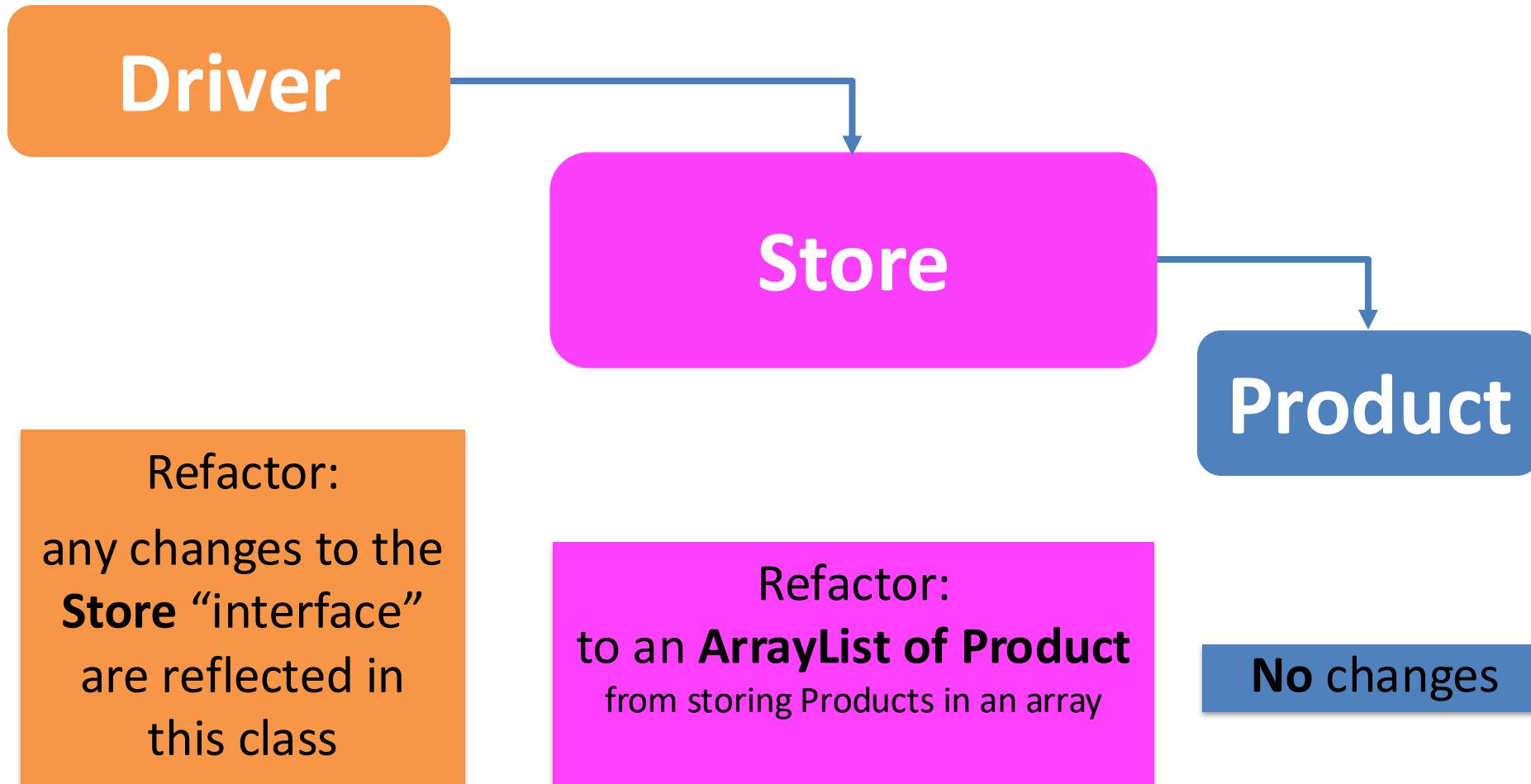
# Shop V3.0

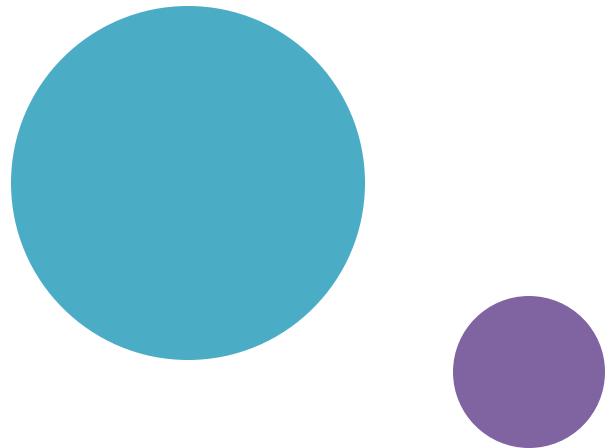


GOAL: to keep all the functionality of V2.2 but  
use an **ArrayList of Products**  
instead of an array.

# Shop V3.0 – changes to classes (refactoring)

---

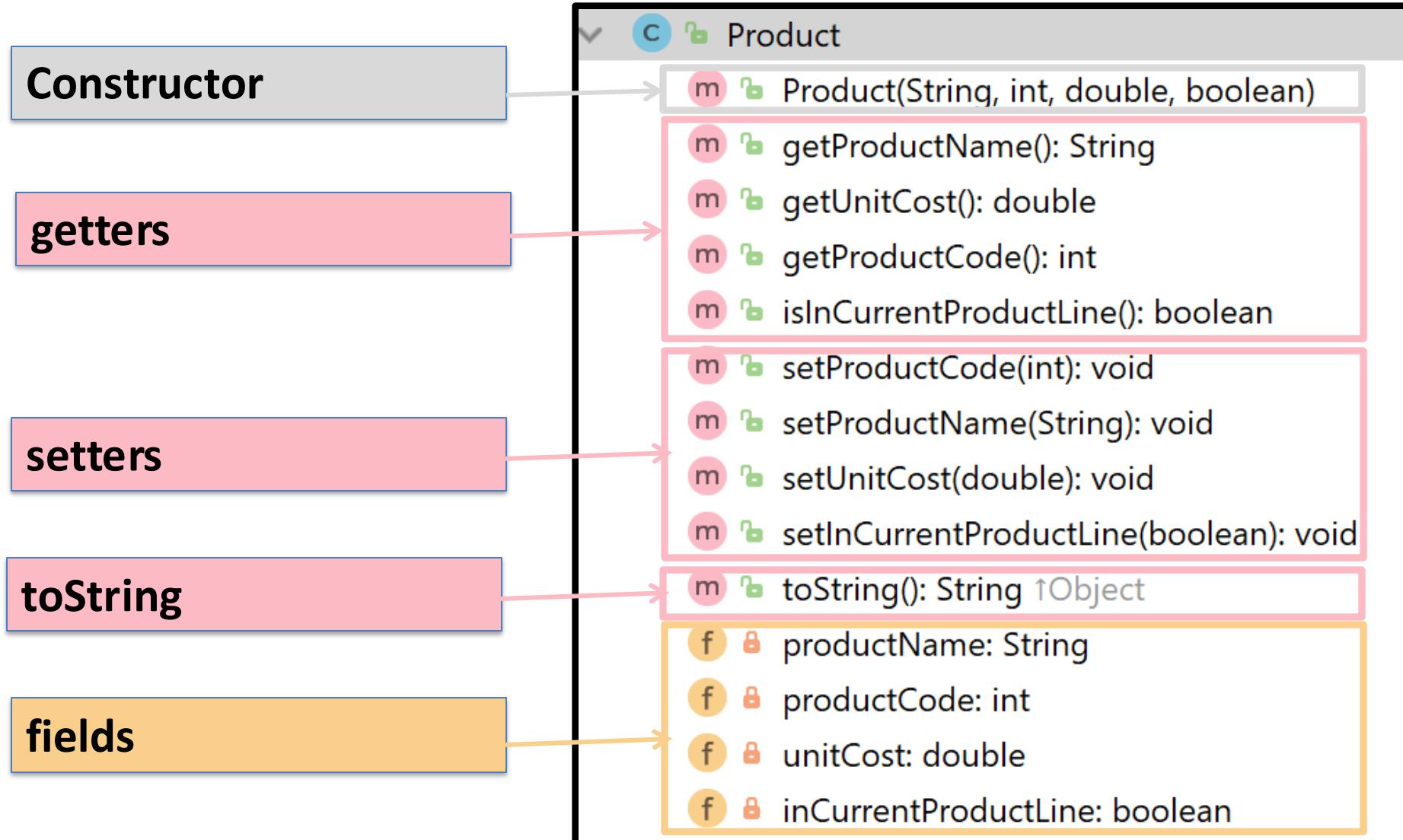




**PRODUCT**

No changes

# Product



# Product

Our Product class contains four fields - instance variables

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

# Product

The **constructor** uses the data passed in the four parameters to update the instance fields.

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

Name Overloading using keyword **this**.

Note : we are omitting validation to simplify the example

# Product

The class has **getters** for each instance field.

```
public String getProductName () {  
    return productName;  
}  
  
public double getUnitCost () {  
    return unitCost;  
}  
  
public int getProductCode () {  
    return productCode;  
}  
  
public boolean isInCurrentProductLine () {  
    return inCurrentProductLine;  
}
```

# Product

The class has **setters** for each instance field.

```
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;  
}
```

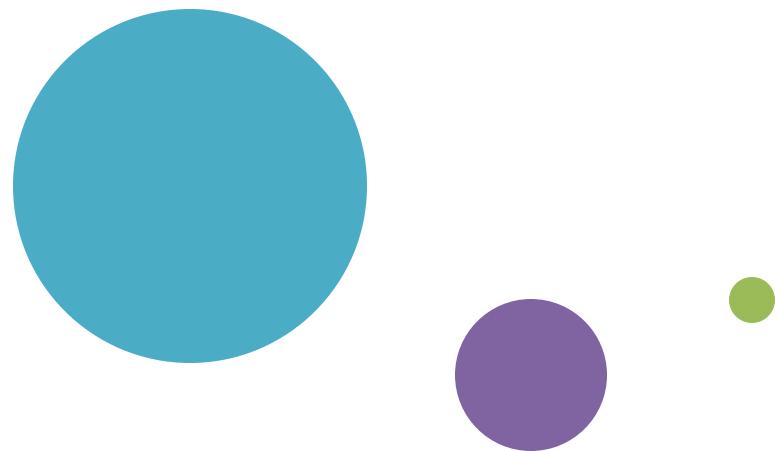
Note : Again, we are omitting validation to simplify the example

# Product

The class has a **toString** method to return a String containing a user-friendly representation of the object state.

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

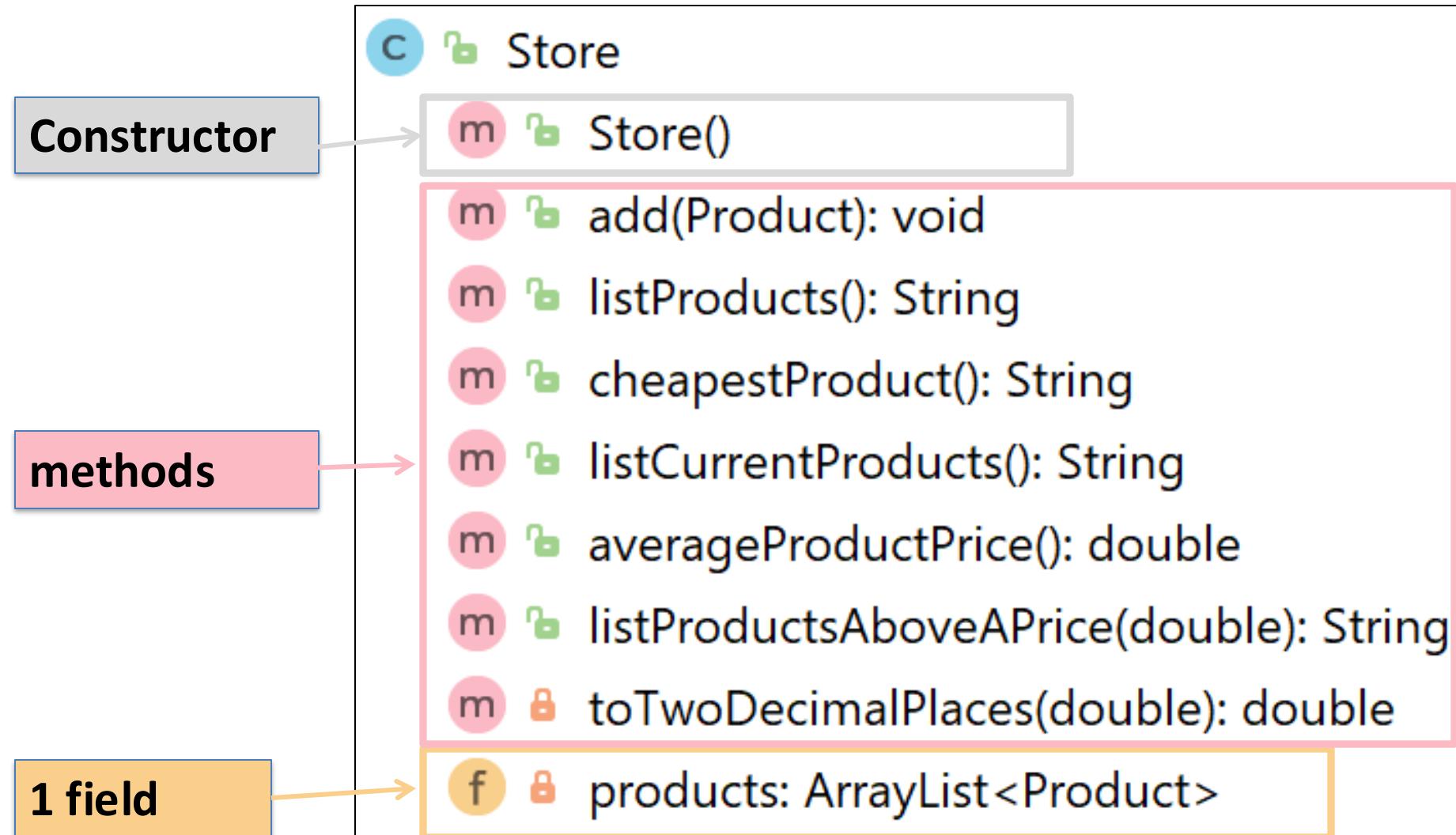
We will call this method from the **Store** class that we will construct over the next few slides.



# STORE

Refactor to an  
**ArrayList of Product**  
from storing  
Products in an array

# Store



# Fields

- The Store class now has just one field called **products**
  - an **ArrayList** of **Product**.



**products: ArrayList<Product>**

## Declaring an **ArrayList** of Product

importing the  
ArrayList class so  
we can use it.

```
import java.util.ArrayList;  
  
public class Store  
{  
    private ArrayList<Product> products;  
  
    // constructor  
    public Store()  
    {  
        products = new ArrayList<Product>();  
    }  
  
}
```

# Declaring an **ArrayList** of Product

importing the ArrayList class so we can use it.

declaring an ArrayList of Product as a private instance variable.

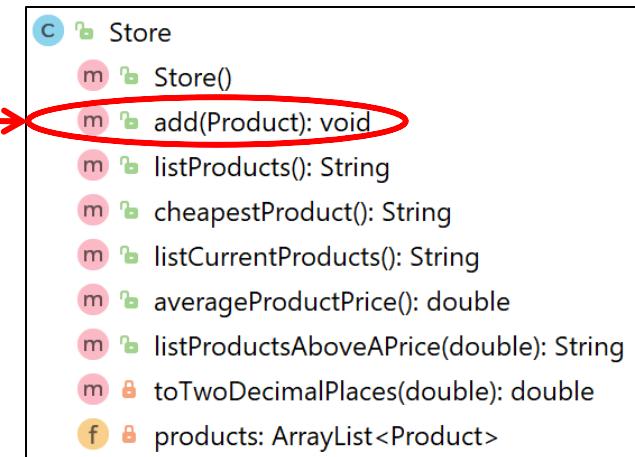
calling the constructor of the ArrayList class to build the ArrayList object.

```
import java.util.ArrayList;  
  
public class Store  
{  
    private ArrayList<Product> products;  
  
    // constructor  
    public Store()  
    {  
        products = new ArrayList<Product>();  
    }  
}
```

# Methods (1)

These methods work on the **ArrayList** to:

1. add Products
2. print out the contents
3. print out the cheapest product



# Add a product object to an ArrayList of Product

Store

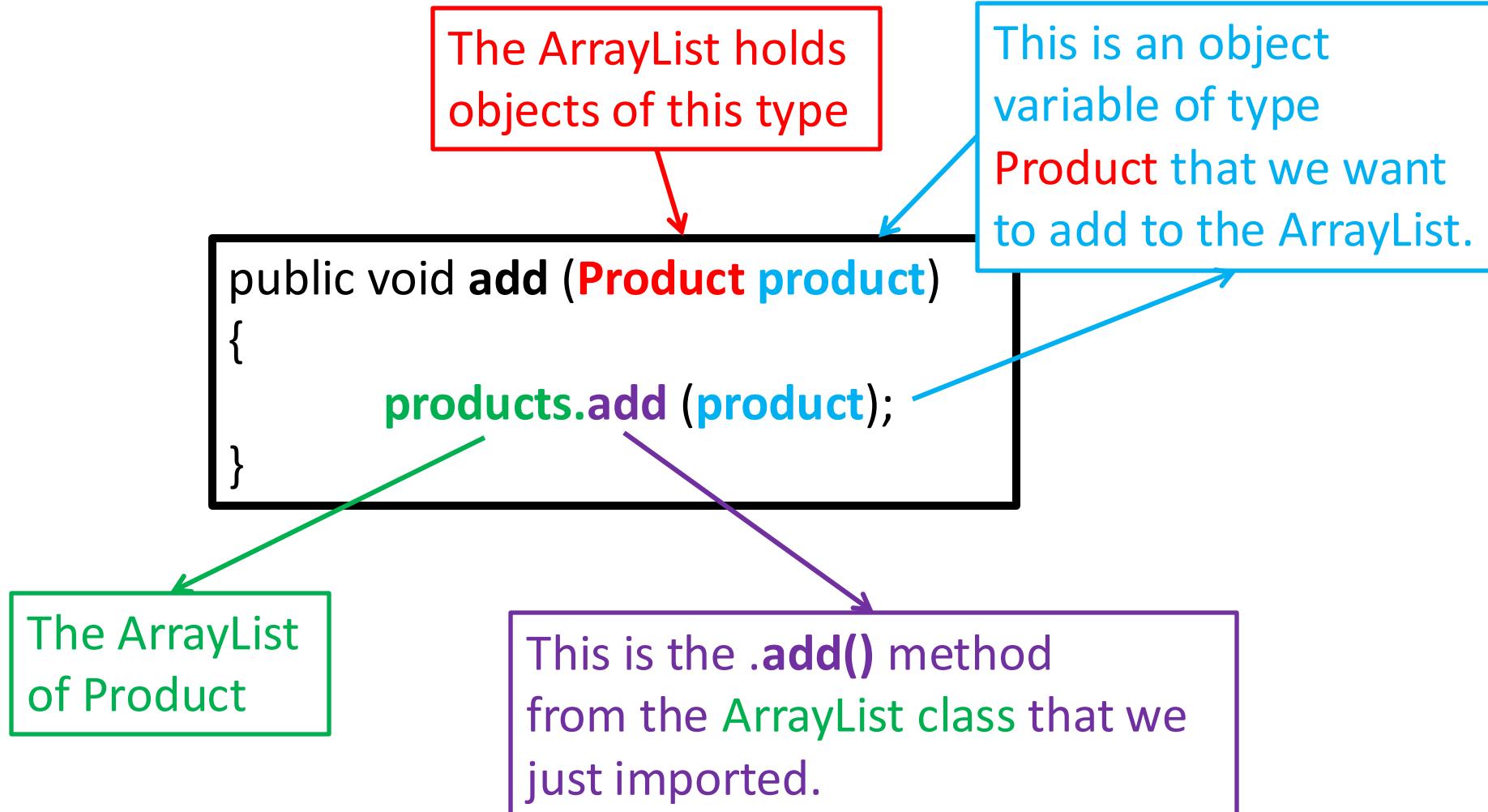
The ArrayList holds  
objects of this type

This is an object  
variable of type  
Product that we want  
to add to the ArrayList.

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

# Add a product object to an ArrayList of Product

Store



# Add a product object to an ArrayList of Product

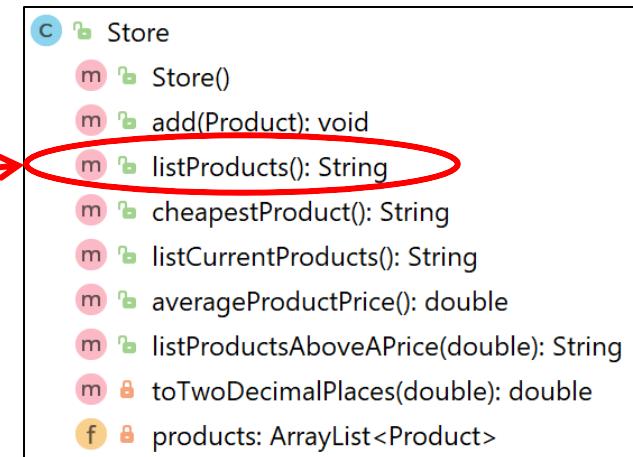
Store

```
import java.util.ArrayList;  
  
public class Store{  
  
    private ArrayList<Product> products;  
  
    public Store(){  
        products = new ArrayList<Product> ();  
    }  
  
    public void add (Product product){  
        products.add (product);  
    }  
}
```

# Methods (2)

These methods work on the **ArrayList** to:

1. add Products
2. print out the contents
3. print out the cheapest product



# Print out the contents

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

## Sample Output

```
0: Product description: Product1, product code: 1, unit cost: 45.99, currently in product line: true  
1: Product description: Product2, product code: 2, unit cost: 12.99, currently in product line: false  
2: Product description: Product3, product code: 3, unit cost: 23.5, currently in product line: true
```

# Store

## Print out the contents

If the size of the products ArrayList is **zero**,  
return the String “No products” to the Driver class to be printed.

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

### Sample Output

```
0: Product description: Product1, product code: 1, unit cost: 45.99, currently in product line: true  
1: Product description: Product2, product code: 2, unit cost: 12.99, currently in product line: false  
2: Product description: Product3, product code: 3, unit cost: 23.5, currently in product line: true
```

## Print out the contents

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

If there are products in the ArrayList...  
return a String containing the index number of each product & the product details.

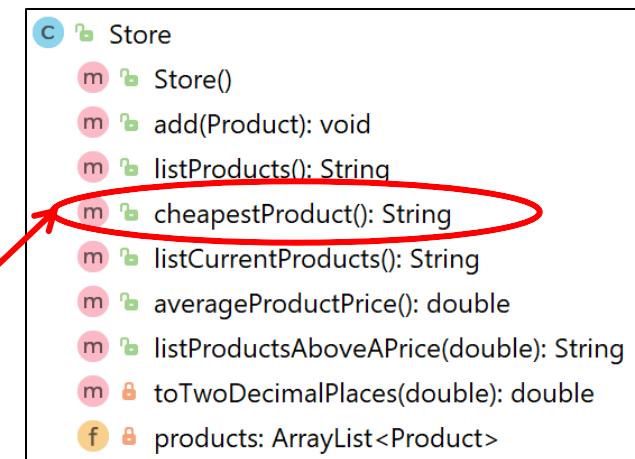
## Sample Output

```
0: Product description: Product1, product code: 1, unit cost: 45.99, currently in product line: true  
1: Product description: Product2, product code: 2, unit cost: 12.99, currently in product line: false  
2: Product description: Product3, product code: 3, unit cost: 23.5, currently in product line: true
```

# Methods (3)

These methods work on the **ArrayList** to:

1. add Products
2. print out the contents
3. print out the cheapest product



# Product Finding the Cheapest Product

getter

private field – unit cost

C	Product
m	Product(String, int, double, boolean)
m	getProductName(): String
m	getUnitCost(): double
m	getProductCode(): int
m	isInCurrentProductLine(): boolean
m	setProductCode(int): void
m	setProductName(String): void
m	setUnitCost(double): void
m	setInCurrentProductLine(boolean): void
m	toString(): String ↑Object
f	productName: String
f	productCode: int
f	unitCost: double
f	inCurrentProductLine: boolean

# Finding the Cheapest Product Algorithm (numbered steps)

1. If products have been added to the ArrayList
  - 1.1 Assume that the first Product in the ArrayList is the cheapest (set a local variable to store this object).
  - 1.2 For all product objects in the ArrayList
    - 1.2.1 if the current product cost is lower than the cost of the product object stored in the local variable,
      - 1.2.1.1 update the local variable to hold the current product object.
  - end if
  - end for
- 1.3 Return the name of the cheapest product.
- else
- 1.4 Return a message indicating that no products exist.
- end if

# Finding the Cheapest Product **(step 1.)**

---

Store

Working on the outer if statement (step 1.)

```
if products have been added to the ArrayList
    return the cheapest product
else
    return a message indicating that no products exist.
end if
```



Q: How do we write the code for this algorithm?



```
if (products.size() != 0) {  
    //return the cheapest product  
}  
else{  
    return "No products are in the ArrayList";  
}
```



Another  
valid  
approach!

```
if (!products.isEmpty()) {  
    //return the cheapest product  
}  
else{  
    return "No products are in the ArrayList";  
}
```

## Working on step 1.1

```
if products have been added to the ArrayList
    // 1.1 Assume that the first Product in the ArrayList is the cheapest
    // (set a local variable to store this object).
else
    return a message indicating that no products exist.
end if
```



Q: How do we write the code for this step?

# step 1.1



```
if (products.size() != 0) {
    Product cheapestProduct = products.get(0);
}
else{
    return "No products are in the ArrayList";
}
```

## Working on the for loop **step 1.2**

```
if products have been added to the ArrayList
    // 1.1 Assume that the first Product in the ArrayList is the cheapest
    // (set a local variable to store this object).
    // 1.2 For all product objects in the ArrayList
    //      determine the cheapest product
    // end for
else
    return a message indicating that no products exist.
end if
```



Q: How do we write the code for this step?

## step 1.2



```
if (products.size() > 0) {  
    Product cheapestProduct = products.get(0);  
    for (Product product : products)  
    {  
    }  
}  
else{  
    return "No products are in the ArrayList";  
}
```

# Store

## for each loop

```
if (products.size() > 0) {  
    Product cheapestProduct = products.get(0);  
    for (Product product : products)  
    {  
    }  
}  
else{  
    return "No products are in the ArrayList";  
}
```

**Product:** This is the type of object that is stored in the ArrayList.

**product:** This is the reference to the current object we are looking at in the ArrayList. As we iterate over each object in the ArrayList, this reference will change to point to the next object, and so on.

**products:** This is the ArrayList of Product.

## step 1.2.1

Store

1. If products have been added to the ArrayList
  - 1.1 Assume that the first Product in the ArrayList is the cheapest (set a local variable to store this object).
  - 1.2 For all product objects in the ArrayList
    - 1.2.1 if the current product cost is lower than the cost of the product object stored in the local variable,**
      - 1.2.1.1 update the local variable to hold the current product object.

**end if**

end for

- 1.3 Return the name of the cheapest product.
- else
- 1.4 Return a message indicating that no products exist.



**Q: How do we write the code for this step?**

## step 1.2.1



```
if (products.size() > 0) {
    Product cheapestProduct = products.get(0);
    for (Product product : products) {
        if (product.getUnitCost() < cheapestProduct.getUnitCost())
            {
            }
    }
}
else
{
    return "No products are in the ArrayList";
}
```

## Step 1.2.1.1

1. If products have been added to the ArrayList
  - 1.1 Assume that the first Product in the ArrayList is the cheapest (set a local variable to store this object).
  - 1.2 For all product objects in the ArrayList
    - 1.2.1 if the current product cost is lower than the cost of the product object stored in the local variable,  
**1.2.1.1 update the local variable to hold the current product object.**  
end if
  - end for
  - 1.3 Return the name of the cheapest product.
  - else
  - 1.4 Return a message indicating that no products exist.
- end if



**Q: How do we write the code for this step?**

## Step 1.2.1.1



```
if (products.size() > 0) {  
    Product cheapestProduct = products.get(0);  
    for (Product product : products) {  
        if (product.getUnitCost() < cheapestProduct.getUnitCost()) {  
            cheapestProduct = product;  
        }  
    }  
}  
else{  
    return "No products are in the ArrayList";  
}
```

## Working on the last **step**, 1.3

1. If products have been added to the ArrayList
  - 1.1 Assume that the first Product in the ArrayList is the cheapest (set a local variable to store this object).
  - 1.2 For all product objects in the ArrayList
    - 1.2.1 if the current product cost is lower than the cost of the product object stored in the local variable,
      - 1.2.1.1 update the local variable to hold the current product object.
  - end if
  - end for
- 1.3 Return the name of the cheapest product.**
- else
- 1.4 Return a message indicating that no products exist.
- end if



**Q: How do we write the code for this step?**

## step, 1.3



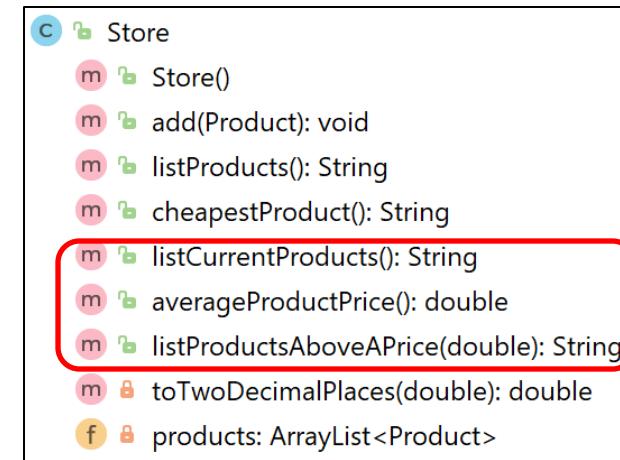
```
if (products.size() > 0){  
    Product cheapestProduct = products.get(0);  
    for (Product product : products){  
        if (product.getUnitCost() < cheapestProduct.getUnitCost()) {  
            cheapestProduct = product;  
        }  
    }  
    return cheapestProduct.getProductName();  
}  
else{  
    return "No products are in the ArrayList";  
}
```

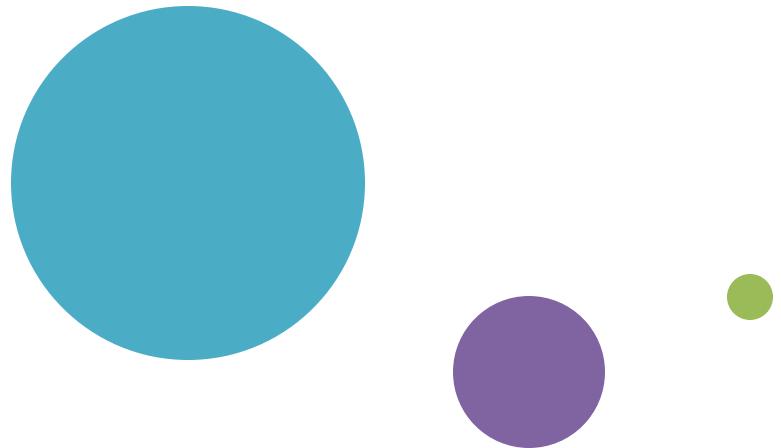
# Methods

We have look at the following methods:

1. add Products
2. print out the contents
3. print out the cheapest product

There are other methods in Store that operate on the ArrayList. They have a similar approach to the methods above; we can look at these in labs if you wish.





# DRIVER

Refactor this class to handle a small change to the **Store** “interface”.

# Store

## Constructor

This was the only change to the public interface; we removed the parameter!

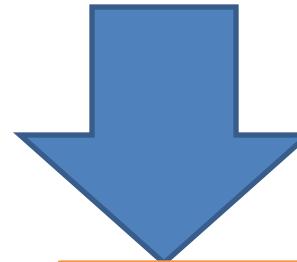
The diagram shows a UML class named 'Store'. It has a constructor 'Store()' and several methods: 'add(Product): void', 'listProducts(): String', 'cheapestProduct(): String', 'listCurrentProducts(): String', 'averageProductPrice(): double', 'listProductsAboveAPrice(double): String', 'toTwoDecimalPlaces(double): double', and a field 'products: ArrayList<Product>'.

- c Store
- m Store()
- m add(Product): void
- m listProducts(): String
- m cheapestProduct(): String
- m listCurrentProducts(): String
- m averageProductPrice(): double
- m listProductsAboveAPrice(double): String
- m toTwoDecimalPlaces(double): double
- f products: ArrayList<Product>

## Driver

Previously our Shop used an array and we needed to know how many Products to store:

```
store = new Store(numberProducts);
```



Now that we are using an ArrayList, we don't need to set a capacity, so our constructor call becomes:

```
store = new Store();
```

# Driver

Our output when we run V3.0 of the app:

```
How many Products would you like to have in your Store? 3
Enter the Product Name: Product 1
Enter the Product Code: 1234
Enter the Unit Cost: 12.99
Is this product in your current line (y/n): y
Enter the Product Name: Product 2
Enter the Product Code: 2345
Enter the Unit Cost: 7.99
Is this product in your current line (y/n): n
Enter the Product Name: Product 3
Enter the Product Code: 6745
Enter the Unit Cost: 49.99
Is this product in your current line (y/n): y
Shop Menu
-----
1) List the Products
2) List the current products
3) Display average product unit cost
4) Display cheapest product
5) List products that are more expensive than a given price
0) Exit
==>>
```

## V3.0 output:

```
How many Products would you like to have in your Store? 3
Enter the Product Name: Product 1
Enter the Product Code: 1234
Enter the Unit Cost: 12.99
Is this product in your current line (y/n): y
Enter the Product Name: Product 2
Enter the Product Code: 2345
Enter the Unit Cost: 7.99
Is this product in your current line (y/n): n
Enter the Product Name: Product 3
Enter the Product Code: 6745
Enter the Unit Cost: 49.99
Is this product in your current line (y/n): y
Shop Menu
-----
1) List the Products
2) List the current products
3) Display average product unit cost
4) Display cheapest product
5) List products that are more expensive than a given price
0) Exit
==>>
```

No difference in output...we only changed the internal storage mechanism!

## V2.2 output:

```
How many Products would you like to have in your Store? 3
Enter the Product Name: Product 1
Enter the Product Code: 1234
Enter the Unit Cost: 12.99
Is this product in your current line (y/n): y
Enter the Product Name: Product 2
Enter the Product Code: 2345
Enter the Unit Cost: 7.99
Is this product in your current line (y/n): n
Enter the Product Name: Product 3
Enter the Product Code: 6745
Enter the Unit Cost: 49.99
Is this product in your current line (y/n): y
Shop Menu
-----
1) List the Products
2) List the current products
3) Display average product unit cost
4) Display cheapest product
5) List products that are more expensive than a given price
0) Exit
==>>
```

# Collections

---

- Allow an **arbitrary number** of objects to be stored.
- Are implemented in **Java's Class libraries** which contain tried-and-tested collection classes.
- In Java's class libraries are called ***packages***.
- We have used the **ArrayList** class from the **`java.util`** package.



# ArrayList

---

- Items may be **added** and **removed**.
- Each item has an **index**.
- **Index values may change** if items are removed (or further items added).
- The main `ArrayList` methods are:
  - `add()`
  - `get()`
  - `remove()`
  - `size()`
- `ArrayList` is a parameterized or generic type.



# Questions?

---

