

# OOP THEORY

Author: Mr H Peuckert

## ALGORITHMS/PSEUDO CODE



To be able to answer anything they throw at you in Section B of the PRACTICAL EXAMINATION & the OOP Question in THEORY EXAMINATION, you need to do the program in these notes practically and study all the theory and definitions.

### Program Theme: Products sold at a Grocery Shop

In the explanation of these notes we will consider the following theme: A grocery shop wants to manage the products they sell. They have stored the data of the products in secondary storage in a text file called **Products.txt**.

Examine the contents of the text file:

```
TOM01#Simba Tomato Chips#125 g#F#16.50#5#31
CHE01#Fasta Pasta Cheese#65 g#F#13.20#3#186
CAN01#B-well Canola Oil#750 ml#F#68.70#6#73
NIV01#Nivea Body Lotion#400 ml#T#37.80#4#19
GAR01#Smash Garlic Butter#104 g#F#21.00#5#93
GIL01#Gillette Shaving Foam#200 ml#T#29.30#4#24
TOM02#Simba Tomato Chips#30 g#F#7.90#3#85
HAN01#Handy Andy Lemon#750 ml#O#33.50#6#49
```

Each line stores the information of one product in the following format:

```
TOM01 # Simba Tomato Chips # 125 g # F # 16.50 # 5 # 31
```

**Product ID # Product Name # Weight of the product (in g or ml) # Type of Product # Cost Price # Markup Percentage # Quantity in Stock**

**Type of Product** is represented by the following characters in the text file:

'F' – Food    'T' – Toiletries    'O' – Other

The **Cost Price** stored in the file does not include VAT and is the purchase price – it has not been marked up yet. Your program will do that by adding the VAT and marking it up by the value stored in the **Markup Percentage** field (when creating a product object).

### QUESTION 1

Use the class diagram below to create a class called **Product**. This class will be used to create objects that will store the details of the products that are sold at the Grocery shop. The diagram below indicates the properties and methods that are required. You must use these names in your program!

You need to be able to create your own class diagram!

#### **Product**

##### **Properties:**

- prodID : String
- prodName : String
- weight : String
- prodType : int
- sellPrice : double
- quantity : int

**NOTE:** When doing this in Java, replace – with the word **private**  
Whenever you see the : character in a class diagram  
you need to take the data type to the front of the  
variable name / method name, e.g.

– prodName : String

becomes private String prodName;

+ (static/class constant) double VAT = 15

**constants.**

+ (static/class constant) integer TYPE FOOD = 1

variables/fields

+ (static/class constant) integer TYPE TOILETRIES = 2

+ (static/class constant) integer TYPE OTHER = 3

**NOTE:** These are called

They are just like

but we assign values to them  
that can never be changed.

##### **Methods:**

– addVATandMarkUp(costPrice : real, markup : real) : real

**NOTE:** This is **Helper Method**  
It is **private**

+ Construct( pi : String, pn : String, w : String, pt : char, costPrice : real, markup : real, quantity : int)

+ getProdID : String

**NOTE:** Here the () are omitted. You need to add the round  
brackets after each method name in Java.

+ getProdName : String

+ getWeight : String

+ getProdType : integer	
+ getSellPrice : real	
+ getQuantity : integer	
+ setProdID( pi : String)	<b>NOTE:</b> Setter methods do not have return types because they
+ setProdName(pn : String)	do not have a return statement in their code.
+ setWeight(w : String)	So there are no : <b>data type</b> at the end.
+ setPrice(cP : real, mU : real)	So remember to add the word <b>void</b> before the method
+ setQuantity(quantity : integer)	name when doing it in Java.
+ getProductTypeName(pt : integer) : String	<b>NOTE:</b> Another <b>Helper Method</b>
+ toString : String	<b>NOTE:</b> Remember to add the round brackets after the method name.

1.1 Write code to create a **new class** called **Product**.

**public class Product {**

 **START A NEW CLASS!**

1.2 Write code to create the 6 **properties** for the **Product** class as indicated in the above class diagram.

**FIELDS/PROPERTIES:**

```
private String prodID;
private String prodName;
private String weight;
private int prodType;
private double sellPrice;
private int quantity;
```

**NOTE:** Other terms they can use for properties/fields: instance variables, attributes

**NOTE:** More on **Access Modifiers:**

When declaring these fields/properties we use the **private access modifier** when the field name is preceded with a **- character** in the class diagram and we use the **public access modifier** when the method name is preceded with a **+ character** in the class diagram.

Do not confuse it with Accessor method. It is the word *private*, *public* or *protected* we use when declaring fields or creating methods, e.g. *private String name;* or *public String getName();* Private means that the field or method can only be used in the class in which it was created (it is only accessible in that class). It cannot be used from other classes. Public means that the field or method can be used in other classes (it is accessible from other classes – it can be used from other classes).

**1.3** Write code to create 4 **constants** for the **Product** class as indicated in the above class diagram.


**CONSTANTS:**

```
public static final double VAT = 15;

public static final int TYPE_FOOD = 1;
public static final int TYPE_TOILETRIES = 2;
public static final int TYPE_OTHER = 3;
```

**NOTE: A “brief” explanation of constants:**

A **constant** is like a variable but its value can never be changed:

int num1 = 19;	← Here we create a <b>variable</b> called num1 and store 19 in it.
num1 = 20; ✓	← Here we change its value to 20 (we assign (store) 20 to it).
 <b>final</b> int num2 = 21;	Here we create a <b>constant</b> called num2 and store 21 in it. Putting the word <b>final</b> in front means that the value cannot be changed. This is the final value that is stored in it.
num2 = 22; ✗	← This instruction will result in an error and will be underlined in red in Netbeans. Constant's values cannot be changed.

We also **capitalise** the identifier name to indicate that it is a constant.

```
final int NUM2 = 19;
```

Let us look at a few **fields** and **constants** from the **Product object class**:

If a variable is used as a field/property at the top of an object class and we want to use it as a constant (we do not want its value to be changed), then we also add the word **static** in front of it. Why?

```
private String prodName;  
private double sellPrice;
```

**NOTE:**

These variables are seen as fields of an object and will be created for every object. The values stored in them can be changed (using setter methods), e.g. *product1.setSellPrice(96.70);*  
If they were public we could use them from another class by using the format **object.field**, e.g.  
*product1.sellPrice = 96.70;*  
but because they are private, we have to use the setter method (as indicated above).

So the instruction **Product product1** (coded in another class) will create the fields for this object:

*product1.prodName*  
*product1.sellingPrice*

So the instruction **Product product2** will create the fields for this object as well:

*product2.prodName*  
*product2.sellPrice*

**Non-static fields**  
belong to each  
object.

**Important:** Because they are non-static (we do not use the word static when declaring them) these fields will be created for every object. All of these fields will be created in memory.

```
public static final double VAT = 15;
```

**NOTE:**

VAT is a constant which will always store 15 which cannot be changed in the program. It is also **static** meaning it belongs to the class. To use them from another class we use the format **Class.constant**, e.g. *Product.VAT* (not **object.constant**)

**So important:** Because they are static they will not be created for every object.

So if the word static was removed from its declaration, then VAT will be created for every object:

*product1.VAT*  
*product2.VAT* **which is unnecessary!**

For each object, VAT will be created in memory that will take up unnecessary space.

**Static fields** belong  
to the class.

See my explanation at the top of the ProductsUI class for clarity

- 1.4 Create a new method called **addVATandMarkUp** that will be used to add the VAT to the cost/purchase price of the product and increase the price (with VAT already added) by the markup percentage. The cost price and markup percentage must be received as parameters. Use the values from the parameters and the VAT constant, and calculate and return the selling price of the product.

#### HELPER METHOD 1:

```
private double addVATandMarkUp(double costPrice, double markUp) {  
    double pricePlusVAT = 0;  
  
    pricePlusVAT = costPrice + costPrice * VAT / 100;           // Add VAT to costPrice  
    pricePlusVAT = pricePlusVAT + pricePlusVAT * markUp / 100; // Increase price by  
    markup %  
  
    DecimalFormat dec = new DecimalFormat("#.##");             // Change the real  
    number  
    pricePlusVAT = Double.parseDouble(dec.format(pricePlusVAT)); // to 2 decimal places  
  
    return pricePlusVAT;  
}
```

**NOTE:** This is a **Helper method**. In this case it will be used to help the constructor method to calculate the selling price of a product. Helper methods are mostly **private** which means that it can only be used in the class where it was created (in this case in the **Product** class). It cannot be called from other classes. **Its main purpose is** to assist the other public methods in the class and normally to perform calculations that the other public methods need.

- 1.5 Write code to create a **constructor method** that will initialize all 6 properties of the **Product**

class. Please note that the **Type of Product** will be received as a character (F, T or O) and your code needs to assign the correct **constant** to the **prodType** field. Also, make a call to the **addVATandMarkUp** method to calculate the selling price and assign it to the **sellPrice** property of the product object.

#### CONSTRUCTOR METHOD:

```

public Product(String pi, String pn, String w, char pt, double costPrice,
               double markUp, int quantity) {

    prodID = pi;
    prodName = pn;
    weight = w;

    if (pt == 'F') {
        prodType = TYPE_FOOD;
    } else if (pt == 'T') {
        prodType = TYPE_TOILETRIES;
    } else {
        prodType = TYPE_OTHER;
    }

    sellPrice = addVATandMarkUp(costPrice, markUp);

```

```

    this.quantity = quantity;
}

```

field name      parameter name

*// Please note that when the parameter name and  
// the object's field name is the same, then you  
// have to put **this.** before the field name.*

**NOTE:** The **constructor method** is used to (1) create a new object and (2) to assign values to

the fields of the object (received as parameters).

Instead of saying “**create**” we can use the word “**instantiate**”, i.e. “to instantiate an object” = “to create an object”. This must not be confused with the word “**initialise**” which means to give a variable a begin value, e.g. sum = 0;

**NOTE:** This **constant** can be used in other classes using the format **Class.contant**, e.g. Product.TYPE\_FOOD and Product.VAT. So instead of using the value of 15 for the VAT in the program we just use Product.VAT. What is the advantages of this? When the vat changes to 16 we do not have to search through hundreds of lines of code and change

15

to 16. We just change it in one place (at the top of the object class were you declared the constant).

Also, instead of using the value of 1 for the Type of Product we can use

Product.TYPE\_FOOD

(you just type in Product, press full stop and choose TYPE\_FOOD from the list).  
Your code just became more user friendlier to other programmers because instead of seeing the value of 1 they see TYPE\_FOOD so they know immediately the type of Product is food. And instead of using the value of 2 we use TYPE\_TOILETRIES which is more understandable.

**1.6** Write code to create **accessor methods** (also called **getter methods**) for all the properties as indicated in the class diagram.

**ACCESSOR\GETTER METHODS:**

```
public String getProdID() {  
    return prodID;  
}  
  
public String getProdName() {  
    return prodName;  
}  
  
public String getWeight() {  
    return weight;  
}  
  
public int getProdType() {  
    return prodType;  
}  
  
public double getSellPrice() {  
    return sellPrice;  
}  
  
public int getQuantity() {  
    return quantity;  
}
```

**NOTE:** A **getter method** (or called an **accessor method**) is used to return the value of a field (property) of an object.

Because the fields/properties of the object is using the **private** access modifier, they cannot be used from other classes. That is the reason why we create getter methods which are using the



**public** access modifier that will simply just return the value of the field. Since these methods are public they can be used from other classes (they are accessible from other classes).

- 1.7 Write code to create **mutator methods** (also called **setter methods**) for all the properties as indicated in the class diagram.

**MUTATOR/SETTER METHODS:**

```
public void setProdID(String pi) {  
    prodID = pi;  
}  
  
public void setProdName(String pn) {  
    prodName = pn;  
}  
  
public void setWeight(String w) {  
    weight = w;  
}  
  
public void setPrice(double cP, double mU) {  
    sellPrice = addVATandMarkUp(cP, mU);  
}  
  
public void setQuantity(int quantity) {  
    this.quantity = quantity;  
}
```

**NOTE:** A **setter method** (or called a **mutator method**) is used to change the value of the field of an object.

Because the fields/properties of the object is using the **private** access modifier, the values stored in them cannot be changed from other classes. That is the reason why we create setter methods which are using the **public** access modifier that will be used to change the value of a field. Since these methods are public they can be used from other classes (they are accessible from other classes). Some of the code (i.e. the parameters & assignment statement in the body of the method) is identical to the code found in the constructor method since both types of methods are used to initialise/change the values of the objects' fields. In a constructor method many fields are changed whereas in a setter method only one field is normally changed at a time.

- 1.8** Create another method called **getProductTypeName** that will be used to return a word ("Food", "Toiletries" or "Other") indicating the type a product. The method must receive an integer that represents the type of product (1 – Food, 2 – Toiletries, 3 – Other). It must then compare the received value to the constants of the class and return the correct word "Food", "Toiletries" or "Other".

**HELPER METHOD 2:**

```
private String getProductTypeName(int pt) {  
    String word = "";  
  
    if (pt == TYPE_FOOD) {  
        word = "Food";  
    } else if (pt == TYPE_TOILETRIES) {  
        word = "Toiletries";  
    } else {  
        word = "Other";  
    }  
  
    return word;  
}
```

**NOTE:** This **helper method** will be used to display the type of product not as a number, e.g. 2, but as more descriptive word, e.g. "Toiletries". It will be called in the toString method when an object's data is combined into a string.

- 1.9** Write code to create a **toString** method which will return a String comprised of the data of a product. You need to combine the data of an object into one string and format it as indicated in the example below:

**Example:**

```
TOM01: Simba Tomato Chips 125 g (Food)  
31 x R19.92 = R617.52
```

**TOSTRING METHOD:**

```

public String toString() {
    String output = "";

    output = "\n" + prodID + ": " + prodName + " " + weight;

    output = output + " (" + getProductTypeName(prodType) + ")";

    output = output + "\n" + quantity + " x R" + sellPrice;

    DecimalFormat dec = new DecimalFormat("#.##");
    output = output + " = R" + Double.parseDouble(dec.format(quantity * sellPrice));

    return output;
}

```

**NOTE:** The **toString method** is used to combine many of the object's field's values into one string and return it. It will be called/used by other classes when they want to display the data of an object.

They are very similar to getter/accessor methods but instead of just returning one field's data, it will return data from many fields (combined into one string).

**NOTE: Calling the toString method:** When calling/using the toString method from other classes, you will use the format **object.toString()**, e.g. *System.out.println( product1.toString());*

You can actually leave the toString out and just use: *System.out.println( product1 );* which means display the object called *product1*. The toString method will then be automatically be called. Both instructions work the same.

**NOTE: What if you call the toString method and it does not exist in the object class?**

Example: You call the toString method from another class, e.g. *System.out.println( product1 );*

or *System.out.println(product1.toString());* BUT the toString method does not exist in the object class (it was not coded in the Product class). What will be displayed then? Go and test it on your completed program that works properly. Comment the toString method out in the Product class (select/highlight it and press **Crtl** and **/** on the keyboard) – so it does not

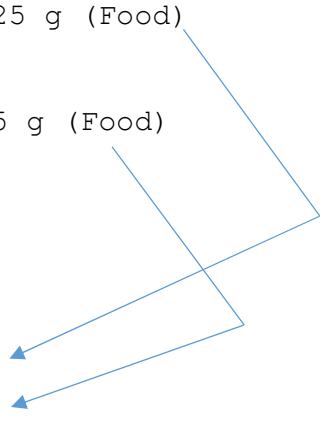
exist anymore. Run the program and see what will be displayed.

Instead of displaying the following:

```
ALL PRODUCTS:  
TOM01: Simba Tomato Chips 125 g (Food)  
31 x R19.92 = R617.52  
CHE01: Fasta Pasta Cheese 65 g (Food)  
186 x R15.64 = R2909.04
```

it will display:

```
ALL PRODUCTS:  
productsui.Product@33909752  
productsui.Product@55f96302
```



**Please note:** When calling a toString method (or displaying an object) when there is no toString method in the object class (which the programmer created), it will call the default

toString method (that can be found in the class called Object). This default toString method in

this Object class will return the hashCode of the object's memory location. It returns the reference to the address space where the object can be found. That reference will then be displayed, e.g. productsui.Product@33909752

By creating our own toString method in our own created object class, the toString method of the Object class will be **overridden** and our toString method will be used instead. A user-defined toString method is needed to combine the data of the object into one string. Here

we

use the following concept:

**Method over-riding:** Where more than one method exists with the same name and the same parameter list.

There are 2 toString methods available in the Products program. The one toString method we coded in the Product class. There also exist a second toString method in the Object

class (that will only be used in the situation when a toString method was never coded in the user-defined object class, e.g. Product). So 2 toString methods, with the same name and the same

parameter list (there are no parameters):

- \* toString() - from Product class

- \* toString() - from Object class

The toString method of the Object class will be **overridden** and our newly created toString method from our newly created object class will be used instead.

This is different from the concept of overloading:

### **Method overloading:**

Where more than one method exists with the same name and a different parameter list

So you have two methods with the exact same name but the parameters are different.

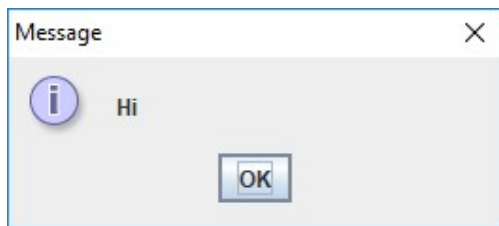
An example of this is the **showMessageDialog** method of the **JOptionPane class**:

There are many of these methods that have the same name but with different parameters:

```
showMessageDialog(Component parentComponent, Object message)
showMessageDialog(Component parentComponent, Object message, String title, int messageType)
```

So calling the method with only 2 arguments will call/use the first method listed above:

```
JOptionPane.showMessageDialog(null, "Hi");
```

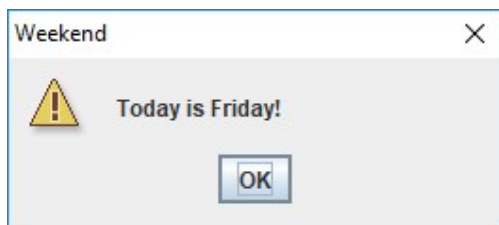


But calling the same method with 4 arguments will call the second method listed above:

```
JOptionPane.showMessageDialog(null, "Today is Friday!",
```

↵

```
"Weekend",JOptionPane.WARNING_MESSAGE);
```



Do you see the **constant** that I used (that is declared at the top of the JOptionPane class):

```
public static final int WARNING_MESSAGE = 2;
```

So instead of using the number 2...

```
JOptionPane.showMessageDialog(null, "Today is Friday!","Weekend",2);
```

...we can use the constant that is static, so **Class.constant**:



## JOptionPane.WARNING\_MESSAGE

We can just type in the class name, press full stop on the keyboard and select it from the list.

**1.10** Create a new method called **getVAT** that will return the value stored in the VAT constant.

### STATIC METHOD:

```
public static double getVAT() {  
    return VAT;  
}
```

**NOTE:** I have to make the method **static** since the VAT constant is also static. We do not actually need this method since VAT is public. We would need it if VAT is private so other classes can get the VAT value. How it is programmed in the memo, we can get the VAT in other classes by saying:

*Product.VAT* or *Product.getVat()*

since they both are **static** ( Class.Field or Class.Method() ) and public:

```
public static final double VAT = 15;  
  
public static double getVAT() {  
    return VAT;  
}
```

## QUESTION 2

**2.1** Write code to create a new class called **ProductsManager**.

**public class ProductsManager {**

 **START A NEW CLASS!**

**2.2** Write code to create the following two **properties** that should not be accessible from outside

the class:

- An array, called **item**, which can be used to store up to 150 **Product** objects.
- An integer counter to keep track of how many products are stored in the **item** array.

### MANAGER CLASS'S FIELDS/PROPERTIES:

```
private Product item [ ] = new Product[150];  
private int size = 0;
```

**NOTE:** The following fields will be created in memory for every object: item[0], item[1], etc...

#### **item[0]**

item[0].prodID	
item[0].prodName	
item[0].weight	
item[0].prodType	
item[0].sellPrice	
item[0].quantity	

#### **item[1]**

item[1].prodID	
item[1].prodName	
item[1].weight	
item[1].prodType	
item[1].sellPrice	
item[1].quantity	

**NOTE:** The code on page 25/26 can be inserted in the user interface class (ProductsUI).

This is more Grade 11 work and explain how objects are created and how the values of their fields can be retrieved in other classes and how the values of their fields can be changed in other classes. The instructions **Product product1** and **Product product2** will only **declare 2 objects** called *product1* and *product2*. **It is better to declare an array** of objects (like what you are doing here in this question). It takes less code to declare many objects, you just use

**Product item[ ] = new Product[150]** and it **will declare 150 objects at once**, called *item[0]*, *item[1]*, *item[2]*, etc. It is also easier to later look at these objects at once. By looping through the array, we can quicker (and with less code) perform calculations and find objects, etc.

**2.3** Write code to create a **constructor method** that will read the contents of the **Products.txt** text file and create an object for each product and store it in the **item** array. The method must

receive a string representing the file name as a parameter and open that file for reading. Each line of text contains data of a product. Read each line from the file and instantiate a Product object and add it into the array. Display a suitable message if the file was not found.

### MANAGER CLASS'S CONSTRUCTOR METHOD:

```

public ProductsManager(String fileName) {

    try {
        Scanner scFile = new Scanner(new File(fileName));

        while (scFile.hasNext()) {
            String line = scFile.nextLine();

            String data[] = line.split("#");

            item[size] = new Product(data[0], data[1], data[2], data[3].charAt(0),
                                     Double.parseDouble(data[4]),
                                     Double.parseDouble(data[5]),
                                     Integer.parseInt(data[6]));

            //TOM01 #Simba Tomato Chips#125 g#F#16.50#5#31
            size++;
        }

    } catch (FileNotFoundException ex) {
        JOptionPane.showMessageDialog(null, "Cannot find Products.txt");
    }

}

```

- 2.4** Write code that will create a method called **getAllProducts**. This method should return a string that contains the information of all the products. Each product should appear on its own line. Use the object's **toString** method that you have created in question 1.9.

**METHOD USED TO DISPLAY ALL DATA (RETURN AS ONE STRING):**

```

public String getAllProducts() {
    String output = "";

```



```

        for (int i = 0; i < size; i++) {
            output = output + item[i].toString() + "\n";
        }
        return output;
    }

```

- 2.5** Write code that will create a method called **totalWorth**. The method must loop through the **item** array and calculate the total worth of all the products in stock. Return the result.

**METHOD TO CALCULATE THE TOTAL WORTH OF ALL THE PRODUCTS IN STOCK:**

```

public double totalWorth() {
    double total = 0;
    for (int i = 0; i < size; i++) {
        total = total + item[i].getQuantity() * item[i].getSellPrice();
    }
    return total;
}

```

- 2.6** Write code that will create a method called **AverageWeight**. Calculate and return the average weight of only the **foods** that are measured in **grams**.

**METHOD TO CALCULATE THE AVERAGE WEIGHT OF FOODS THAT ARE MEASURED IN GRAMS:**

```

public double AverageWeight() {
    double total = 0;
    int count = 0;
    for (int i = 0; i < size; i++) {

```

```

        if (item[i].getProdType() == Product.TYPE_FOOD) {
            String data[] = item[i].getWeight().split(" "); // "125 g" will be split around the space
                                                            // data[0]: "125"  data[1]: "g"

            double weight = Double.parseDouble(data[0]); // weight: 125.0
            String unit = data[1];                       // unit: "g"
            if (unit.equals("g")) {
                total = total + weight;
                count++;
            }

        } //end if
    } // end for loop

    double avg = total / count;

    return avg;
}

```

- 2.7** Write code that will create a method called **getProductQuantity** that must receive a String representing the ID of a product as a parameter. Find this product in the **item** array and return the number of that specific product in stock (it's quantity). Stop searching as soon as the product has been found.

#### **METHOD TO RETURN THE QUANTITY OF A CERTAIN PRODUCT:**

```

public int getProductQuantity(String id) {
    int qty = 0;

    for (int i = 0; i < size; i++) {

        String prodIDNumber = item[i].getProdID();

        if (prodIDNumber.equals(id)) {

```

```

        qty = item[i].getQuantity();
        i = size;           //So that it will stop looping when the product was found
    } // end if
} // end for

return qty;
}

```

- 2.8** Write code that will create a method called **decreaseQuantity** that must receive a String and an integer as parameters representing the ID of a product and the amount by which its quantity needs to decrease by. Find this product in the **item** array and decrease its quantity by the value received as a parameter. Again, stop searching as soon as the product has been found. This method will be used when someone buys an item and the object's quantity field needs to be updated.

### METHOD TO DECREASE THE QUANTITY OF A PRODUCT IN STOCK BY A CERTAIN AMOUNT:

```

public void decreaseQuantity(String id, int byAmount) {

    for (int i = 0; i < size; i++) {

        String prodIDNumber = item[i].getProdID();

        if (prodIDNumber.equals(id)) {
            int currentQty = item[i].getQuantity();
            int newQty = currentQty - byAmount;
            if (newQty >= 0) {

```

```

        item[i].setQuantity(newQty);
    }
    i = size;           //So that it will stop looping when the product was found
} // end if

} // end for

}

```

### QUESTION 3

- 3.1 Write code to create a simple user interface called **ProductsUI** which will declare and instantiate a **ProductsManager** object (send the **Products.txt** file to the **ProductsManager**'s constructor method).

**public class ProductsUI {**  **START A NEW CLASS!**

```

    public static void main(String[] args) {

```

```

        ProductsManager pm = new ProductsManager("Products.txt");

```

- 3.2 Write code that will display the following by calling the appropriate methods in the **ProductsManager** class. You must call the methods in the following order:
- Display all the products in stock.
  - Display the total worth of all the products in stock.
  - Display the average weight of foods that are measured in gram.
  - Display the quantity of 30 g tomato chips in stock (Product ID: TOM02)
  - Ten packets of 30 g tomato chips were sold. Update the quantity in the array.
  - Again, display the quantity of 30 g tomato chips in stock.

```

System.out.println("\n\nALL PRODUCTS: \n" + pm.getAllProducts());

```

```

System.out.println("\nTOTAL WORTH OF ALL THE PRODUCTS IN STOCK: R"
    + pm.totalWorth());

```

```
System.out.println("\n\nAVERAGE WEIGHT OF FOODS THAT ARE MEASURED IN GRAM:
```

```
"
```

```
+ pm.AverageWeight() + " g");
```

```
System.out.println("\n\nQUANTITY OF TOMATO CHIPS IN STOCK: "
```

```
+ pm.getProductQuantity("TOM02"));
```

```
System.out.println("10 PACKETS OF TOMATO CHIPS SOLD");
```

```
pm.decreaseQuantity("TOM02", 10);
```

```
System.out.println("NEW QUANTITY OF TOMATO CHIPS IN STOCK: "
```

```
+ pm.getProductQuantity("TOM02"));
```

## DEFINITIONS: OOP

- **OOP Definition:**

Object-oriented programming (OOP) refers to a type of computer programming (software design) in which programmers define not only the data **type of a data structure**, but also the

types of **operations (functions)** that can be applied to the data structure.

**methods**

**fields/properties**

- **Advantages of using Object Orientated Programming:**

- **Code reusability:**

Objects can be reused within and across applications.

- **Improved software maintainability:**

It makes software easier to maintain. Since the design is modular, part of the system can be updated in case of issues without a need to make large-scale changes. Objects can be maintained separately, thus locating and fixing problems more easily

- **Faster development:**

Object-oriented programming languages come with rich libraries of objects, and code developed during projects is also reusable in future projects.

- **Class/Fields/Methods:**

A class consist of fields and methods:

- A **class** is a blueprint/plan for an object.
- A **field** (also known as the property/attribute) of an object stores a characteristic of an object, e.g. the *colour* field created of a *car* object will store the colour of a car, e.g. blue.
- A **method** is a set of instructions to perform a task/action.

- **Constructor method:**

The constructor method is used to create a new object and to assign values to the fields of the object (received as parameters).

- **Getter (Accessor) method:**

A getter method (or called an accessor method) is used to return the value of a field of an object.

- **Setter (Mutator) method:**

A setter method (or called a mutator method) is used to change the value of the field of an object.

- **Access modifier:**

*Do not confuse it with Accessor method.* It is the word *private*, *public* or *protected* we use when declaring fields or creating methods, e.g. *private String name*; or *public String getName()*;

Private means that the field or method can only be used in the class in which it was created (it is only accessible in that class). It cannot be used from other classes.

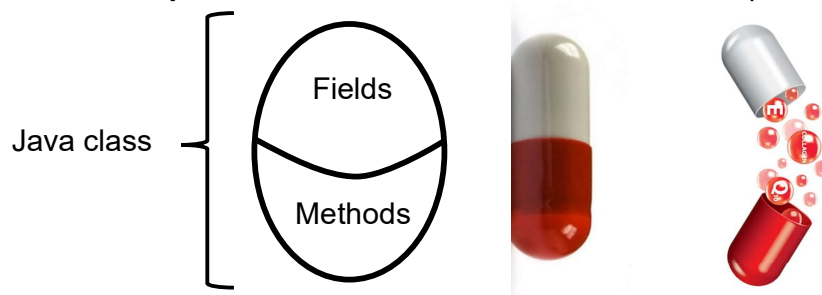
Public means that the field or method can be used in other classes (it is accessible from other classes – it can be used from other classes).

- **Helper method:**

This is a method that normally uses the access modifier *private* and can only be used in the class where it was created. It will only be used by other methods in the same class and will not be used by other classes (maybe other classes will not have a need for it). The Helper method is used in other public methods of the same class to perform a calculation that the public method needs.

- **Encapsulation:**

- A class **encapsulates** the fields and the methods in a capsule-type entity.



**Encapsulation** occurs when a class combines the fields and methods into one unit. When you instantiate an object all the fields are created and the methods have access to these fields.

- A method of protecting data from unwanted access or alteration by packaging it in an object where it is only accessible through the object's interface.
- The hiding of the internal presentation of objects and implementation details from the client program.
- **In other words:** making your fields private, and controlling access to them by using methods (e.g. getter and setter methods).

- **Instantiation:**

*Instantiation* means that we are creating an object of a class.

Let us say we create 2 objects of the Product class called *product1* and *product2*:

```
Product product1 = new Product("TOM03", "All Gold Tomato Sauce", "500 ml", 'F', 46.70, 5, 18);
```

```
Product product2 = new Product("TOM04", "All Gold Tomato Sauce", "250 ml", 'F', 24.80, 3, 27);
```

Then we say we created 2 *instances* of the *Product* class. The object *product1* is an instance of the *Product* class. The object *product2* is another instance of the *Product* class. Each

instance use the same fields and methods of the *Product* class but one instance's fields contain different data than another instance of the class.

In programming, instantiation is the creation of a real instance of a class of objects.

To instantiate is to create such an instance by, for example, defining one particular variation of object within a class, giving it a name, and locating it in some physical place.

- **Parameter:**

- A variable passed to a method.
- Allows the same method to produce different results.
- It is declared in a method header.

- **Static methods:**

- Use the keyword static in the header of the method.
- To call a static method you use the format *Class.method()*

- **Static Data Structures vs Dynamic Data Structures:**

- A static data structure is an organization or collection of data in memory that is fixed in size. This results in the maximum size needing to be known in advance, as memory cannot be reallocated at a later point. Arrays are a prominent example of a static data structure.
- Static data structures stand in contrast to dynamic data structures, wherein with the latter the size of the structure can dynamically grow or shrink in size as needed, which provides a programmer with the ability to control exactly how much memory is utilized.

- **Non Static methods:**

- Does not use the keyword static in the header of the method.



- To call a non-static method you use the format *object.method()*; So you first have to instantiate (create) an object of the class and then call the method by using the object.

```
// THE FOLLOWING INSTRUCTIONS IS NOT PART OF THE PROGRAM
// AND CAN BE INSERTED IN THE USER INTERFACE CLASS FOR EXPLANATION
// PURPOSES:
```

```
// Here we are creating 2 objects called product1 and product2, displaying the values of some
```

```
// of its fields and changing others. The value of the VAT constant is also displayed.
```

```
Product product1 = new Product("TOM03", "All Gold Tomato Sauce", "500 ml", 'F', 46.70, 5, 18);
```

```
System.out.println( "The product1 object contains the following data: \n"
    + product1.toString() ); // OR just: + product1
```

```
System.out.println("\n" + Product.getVAT() + "% VAT has been added to the purchase
    of price of R46.70");
```

```
System.out.println("Yes, that is right, it has been increased by a massive " + Product.VAT
    + "%");
```

```
System.out.println("Further, it's cost has been marked up by 5% to R" +
    product1.getSellPrice());
```

```
int currentQty = product1.getQuantity();
```

```
System.out.println("\nThe number of " + product1.getProdName() + " bottles in stock: "
    + currentQty);
```

```
product1.setQuantity(currentQty - 2);
```

```
System.out.println("2 bottles have been purchased. There are "
```

```

        + product1.getQuantity() + " bottles
        left.");

    Product product2 = new Product("TOM04", "All Gold Tomato Sauce", "250 ml", 'F', 24.80, 3,
27);

    System.out.println("The product2 object contains the following data: \n"
        + product1.toString()); // OR just: + product1

    System.out.println("\n" + Product.getVAT() + "% VAT has been added to the purchase price
        of R24.80");

    System.out.println("Yes, that is right, it has been increased by a massive " + Product.VAT
        + "%");

    currentQty = product2.getQuantity();

    System.out.println("\nThe number of " + product2.getProdName() + " bottles in stock: "
        + currentQty);

```

// PLEASE NOTE that the constant VAT is static and belongs to the class and is not created for every object product1 and product2 unnecessarily. The fields of the objects, e.g prodName, is not-static, so they will be created for every object: product1.prodName and product2.prodName (which is correct).

Now imagine you removed the word static from the declaration of VAT, then it will be created for every object: product1.VAT and product2.VAT which is unnecessary and wastes memory space.

## ALGORITHMS: PSEUDO CODE

An algorithm is the steps we need to follow to solve a problem. Each step represents one line of code in a program. One line of pseudo code can be translated to one instruction in any programming language, e.g. Java, C++, Python, etc.

So someone can write the steps to solve a problem on paper (or type it out in a word processor) instead of writing down the code and a programmer (using any kind of programming language) can then convert each step into an instruction in his/her program. We call this code (steps of solving the problem) Pseudo Code because it is not real code.

In the matric final examination you will have to give solutions to problems by writing down the steps instead of providing the code.

So instead of writing down

```
name = JOptionPane.showInputDialog("Please enter your name");
```

you simply write down

*Enter name*

JAVA	PSEUDO CODE
<b><u>DECLARATIONS:</u></b>	
int age;	<b><u>declare</u> age <u>as</u> integer</b>
String name;	<b><u>declare</u> name <u>as</u> string</b>
boolean found;	<b><u>declare</u> found <u>as</u> boolean</b>
double amount;	<b><u>declare</u> amount <u>as</u> double</b> (or <u>as</u> a real number)
Prisoner convict;	<b><u>declare</u> convict <u>as</u> prisoner</b> (or declare a prisoner object called convict)
<b><u>OUTPUT:</u></b>	

JOptionPane.showMessageDialog(null,"Age:" + age); System.out.println("Age:" + age);	<b><u>display</u> "age:" + age</b>
<b><u>INPUTS:</u></b>  name = JOptionPane.showInputDialog("Enter name");  int age = Integer.parseInt(JOptionPane.showInputDialog("Enter age"));	<b><u>display</u> "enter name"</b> <b><u>input</u> name</b>  <b><u>declare</u> age <u>as</u> integer</b> <b><u>display</u> "enter age"</b> <b><u>input</u> age</b>
<b><u>ASSIGNMENT STATEMENTS / CALCULATIONS:</u></b>  amount = 300;  total = total + 5;  found = true;  count++;  convict = new Prisoner("Thomas Corke",30,'B');	<b>amount <math>\leftarrow</math> 300</b>  <b>total <math>\leftarrow</math> total + 5</b>  <b>found <math>\leftarrow</math> true</b>  <b>add 1 to count</b>  <b>instantiate the object convict from the prisoner class</b>  <i>(in other words: create the object convict from the prisoner class)</i>
<b><u>DECISIONS – IF:</u></b>  if (name.equals("Tony") && age == 21)  if (age != 21)  if (convict[i].getMonth() > 30)	<b><u>if</u> name = "Tony" <u>and</u> age = 21</b>  <b><u>if</u> age <math>\neq</math> 21</b>  <b><u>if</u> convict[i].getMonth() &gt; 30</b>


<p>if (convict[i].getName().equalsIgnoreCase(searchName))</p> <p>if (found == true)                      <u>OR just:</u> if (found)</p> <p>if (found == false)                      <u>OR just:</u> if (!found)</p>	<p><b><u>if</u> convict[i].getName() == seachName</b></p> <p><b><u>if</u> found = true                      <u>OR:</u> if found</b></p> <p><b><u>if</u> found = false                      <u>OR:</u> if not found</b></p>
<p><b><u>LOOPS – FOR:</u></b></p> <p>for (int i = 1; i &lt;= 5; i++)</p> <p style="text-align: center;"><u>OR</u></p> <p>for (int i = 0; i &lt; 5; i++)</p> <p><u>For sorting:</u></p> <p>for (int i = 0; i &lt; size-1; i++)              for (int j = i+1; j &lt; size; j++)</p>	<p><b><u>for</u> i ← 1 <u>to</u> 5</b></p> <p style="text-align: center;"><u>OR</u></p> <p><b><u>for</u> i ← 0 <u>to</u> 4</b></p> <p><u>For sorting:</u></p> <p><b><u>for</u> i ← 0 <u>to</u> size – 1 – 1</b>              <b><u>for</u> j ← i+1 <u>to</u> size – 1</b></p> <p><i>The instructions inside the loop is indented:</i></p> <p style="margin-left: 100px;">for i ← 0 to 5                input name[i]                input age[i]</p>
<p><b><u>LOOPS – WHILE:</u></b></p> <p>while (number != 0)</p> <p><u>Example:</u></p> <p>int age = Integer.parseInt(JOptionPane.showInputDialog("Enter age");</p> <p>while (age &gt; 100) {              JOptionPane.showMessageDialog(null,"Error!");              age = Integer.parseInt(JOptionPane.showInputDialog("Enter age");</p>	<p><b><u>while</u> number ≠ 0</b></p> <p style="margin-left: 100px;">display "Enter age"              input age              while age &gt; 100                    display "Error!"                    display "Enter age"</p>

<pre>Dialog("Enter age"); }</pre>	<pre>input age</pre>
<pre>JOptionPane.showMessageDialog(null,"Age:" + age);</pre>	<pre>display "Age:" + age</pre>

### Algorithms (discussed during IEB conference)

*New or noteworthy*

- *Complete a partial given algorithm*
- *Diagnose errors in an algorithm*
- *Complete a trace table to diagnose errors*
- *Read an article to set a scenario*
- *Give a justified opinion*
- *Slightly more open-ended questions*



*Algorithms*

- *Searching for max/min: always use the first element of the array as your test, not high/low values*
- *Reason:*
  - *What if all numbers are negative?*
  - *What if the array is empty?*
  - *It's all about data integrity/validation*

➤ *Please, please please - no high level languages*

```

i ← 0
flag ← false
size ← size of valArr
pos ← -1
while ( i < size AND flag = false )
    if (inValue = valArr[i])
        flag ← true
        pos ← i
    end if
    i ← i + 1
end while
return pos

```

Question 1: Write down the algorithm in pseudocode to determine the highest number.

int number[] = new int[3];      Declare the number as an array of type integer with size 3

number[0] = -10;      number[0] ← - 10

number [1] = -3;      number[1] ← - 3

number [2] = -13;      number[2] ← - 13

Declare max as an integer

int max = number [0];      max ← number[0]

```

for (int i = 1; i < 3; i++) {
    if (number [i] > max ) {
        max = number [i];
    }
}

```

for i from 1 to 3  
 if (number[i] > max)  
     max ← number[i]  
 end if  
end for

System.out.println("Highest number:" + max);      Display "Highest number:" + max

Question 2: Complete the trace table for the above program.

number[0]	number[1]	number[2]	max	i	i < 3	number[i]>max	output


**Answer:**

number[0]	number[1]	number[2]	max	i	i < 3	number[i]>max	output
-10							
	-3						
		-13					
			-10				
				1			
					True		
						True	
			-3				
				2			



					True		
						False	
				3			
					False		
							Highest number: -3

See what happens if

int max = number [0];

changes to

**max**

int max = 0;

**If all the numbers are negative, the highest number will be 0  
which is wrong! So rather put the first element in the array in**


**Answer:**

-10							
	-3						
		-13					
			0				
				0			
					True		
						False	
				1			
					True		
						False	
				2			
					True		
						False	
				3			
					False		
							Highest number: 0