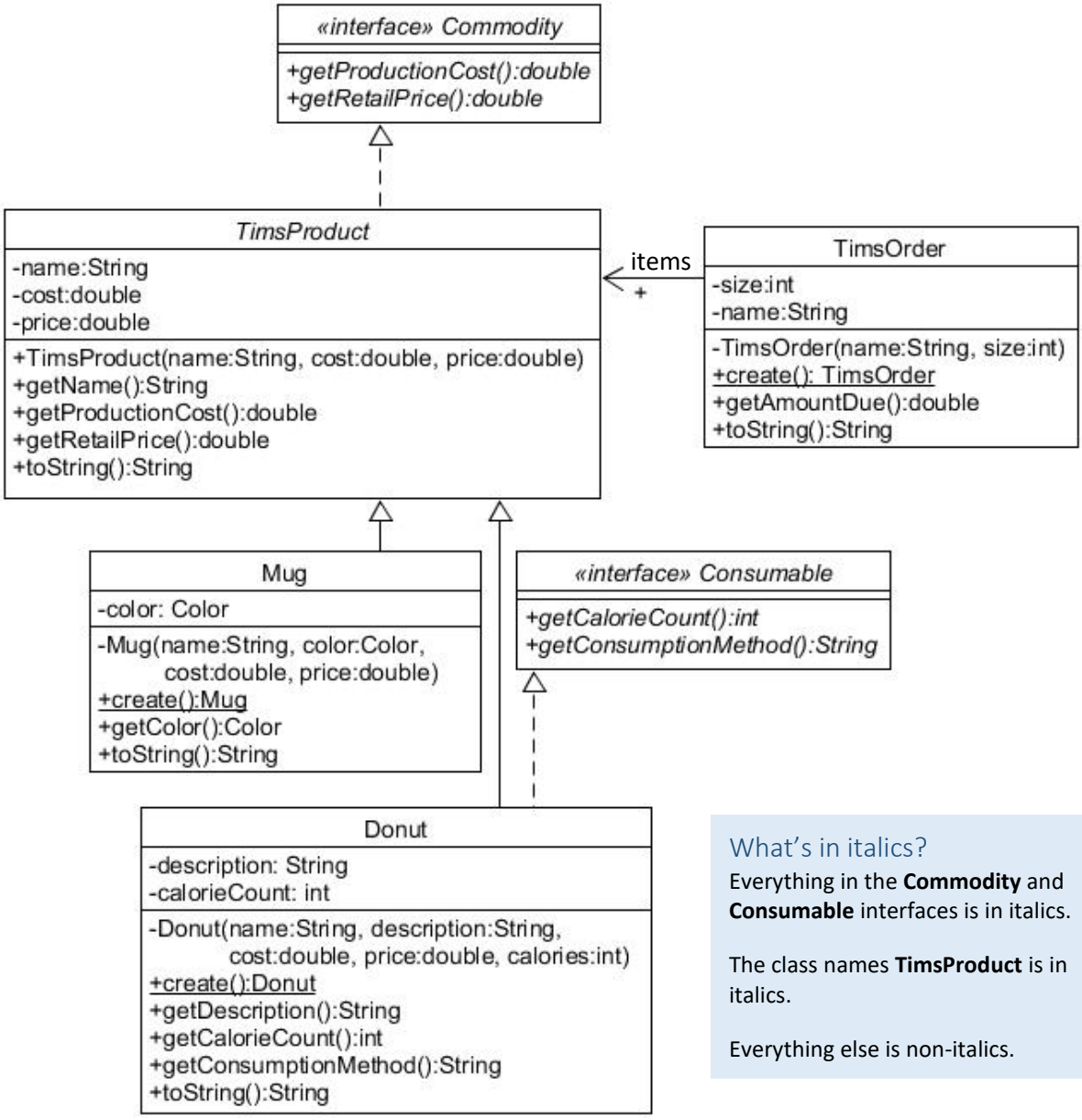
COMP10062: Assignment 7

The Assignment

This assignment is about polymorphism, abstract classes, and interfaces. The UML class diagram below shows an arrangement of classes with inheritance, association, and implementation relationships between them. Your job is to implement the classes and relationships shown here.



What's in italics?

Everything in the **Commodity** and **Consumable** interfaces is in italics.

The class names **TimsProduct** is in italics.

Everything else is non-italics.

Description

The classes in the diagram represent products that are available at Tim's. Some of them are **Consumable** and some are not. Each one is a **TimsProduct** and a **Commodity**. *You must add one more* **Consumable** *product and one more that is not a* **Consumable***.*

Also represented is **TimsOrder**. This class has an array that contains 1 or more **TimsProducts**, representing a named customer's order. The **+** on the association arrow means "1 or more".

Implementation Notes

getConsumptionMethod()

The **getConsumptionMethod()** method in **Consumable** should return a **String** like "Drink it" or "Eat it" as appropriate. It tells you how the product is consumed.

Constructors vs. Create Methods[1]

Notice that many of the constructors are private. The classes with private constructors have static **create()** methods. Instead of calling a constructor, call the **create()** method, like this:

```
Mug m = Mug.create();
```

This method will have a dialog with the user to collect information about the object (in this case, color, cost and price), and will then create and return an object by calling the private constructor. When you add your new products, you should follow this same pattern.

The **TimsOrder** class also uses a **create()** method. This method asks the user their name and how many products they want, then creates a **TimsOrder** object by calling its private constructor. Then for each item in the array, it asks what type the customer wants (**Mug**, **Donut** or one of the other two types you will add), then calls the appropriate **create()** method and stores the result in the array.

Model vs. View

Note that because the **create()** methods are talking to the user, this assignment does not maintain the standard model/view separation. However, you can think of the static methods as implementing the **view**, and the instance methods and variables as implementing the **model**.

getAmountDue()

The **getAmountDue()** method adds up all the retail prices of the products in the order and returns the sum. The **toString()** method returns a **String** with the name of the customer and the **toString()** values of all the products in the order, something like this:

```
Order for: Sam Scott
TimsProduct{name=Official Tim's Mug, cost=1.99, price=19.99}
   Type... Mug{color=0xff0000ff}
TimsProduct{name=Sticky, gooey, good., cost=0.25, price=0.99}
   Type... Donut{description=Kruller, calorieCount=320}
```

---

[1] This is known in software engineering as the **Factory Design Pattern**. The create methods are called **static factory methods**. The Java FX `Color` class uses static factory methods for creating colors (`rgb()`, `web()`, etc.). Each of these methods interprets its arguments and then calls a `Color` constructor and returns the result.

Notice that the **Mug** and **Donut toString()** methods are incorporating the **Mug** and **Donut toString()** methods. You should do this as well, but feel free to make your output look nicer than this!

TestClass.java

Use the code below to test your classes. This short program should initiate an elaborate dialog with the customer through calls to the **create()** methods, then output the order and the total, rounded to two decimal places. You should be able to use this code without changing it.

```
public class TestClass {
    public static void main(String[] args) {
        TimsOrder t = TimsOrder.create();
        System.out.println(t);
        System.out.printf("Total Price: $%.2f\n", t.getAmountDue());
    }
}
```

Advice

1. Go top down. Start with the highest interfaces and classes and work downwards.
2. Make instance variables, and easy methods. For the more complicated methods, just create "stubs" for now.
3. A method stub is an empty method. You make this first and fill in the details later. An example of a method stub for **TimsOrder.create()** is shown below.
4. Once you have the entire structure created like this, you can start filling in the details of each method.

```
public static TimsOrder create() {
    // TODO: Fill in this stub to have a dialog with the user
    //and create a TimsOrder.
    return null;
}
```

Handing In

See the due date in Canvas. Hand in by attaching a zipped version of your **.java** (not .class) files to the Canvas Assignment. *Although the assignment may not be due before the test, you are strongly encouraged to finish as much as you can to help you on the test.*

Make sure you follow the **Documentation Standards** for the course.

Evaluation

Your assignment will be evaluated for performance (40%), structure (40%), and documentation (20%) using the rubric in the drop box.