CMP1903 Object-Oriented Programming
Workshop 8: eCommerce Platform

A client is managing an e-commerce platform that offers different types of products. There is a base Product class that holds general properties such as ProductID, ProductName, and Price. The client has a new requirement to add a new product type called DiscountedProduct, which includes a discount percentage in addition to the regular price. The client also wants a feature to calculate the discounted price based on this percentage.

The client has two types of products:

- **Normal Product**: This product only has a Price and a ProductName.
- **Discounted Product**: In addition to Price and ProductName, it also includes a DiscountPercentage and should have a method to calculate the discounted price.

You have been provided with an initial client implementation, which covers the details below. You are required to understand the code as it is in order to implement the client's requested changes. The relevant .cs files are available on Blackboard.

- **Create a Product Interface (IProduct.cs):** The interface should include methods for getting the ProductName, Price, and CalculateFinalPrice.
- **Create a DiscountedProduct Class (DiscountedProduct.cs):** This class should implement the Product interface, including the DiscountPercentage field. The CalculateFinalPrice method should take the discount into account and return the final price.
- **Ensure that the NormalProduct Class Implements the Product Interface:** The NormalProduct class should implement the Product interface, but the CalculateFinalPrice method should return the normal price without any discount.

Tasks:

Refactoring to a Base Class

The client realizes that many product properties are common and should be part of a base class to promote code reusability and reduce duplication. The Product interface should be simplified to include only behavior, while shared properties like ProductName and Price should be moved to a new ProductBase class to follow the DRY (Don't Repeat Yourself) principle. Additionally, every derived class should be forced to implement the methods CalculateFinalPrice(), GetProductName(), and GetPrice() to ensure that all subclasses provide their own specific implementations for these behaviors.

- **Create a ProductBase Class:** The ProductBase class should contain common properties (ProductName and Price) and methods for getting the product details (GetProductName() and GetPrice()).
- **Ensure Both NormalProduct and DiscountedProduct Inherit from ProductBase:** Both NormalProduct and DiscountedProduct should inherit from ProductBase but still implement the Product interface. This ensures all product types can be treated uniformly via the Product interface, while still allowing for specific behaviors in each derived class.
- **Enforce Implementation of CalculateFinalPrice():** The CalculateFinalPrice() method should be declared as an abstract method in the ProductBase class, ensuring that all derived classes implement it. This enforces consistency across all product types and ensures that each class fulfills the contract defined by the interface.

Common Reporting Method

Now, the client wants to introduce a method to report on products, where the NormalProduct class simply displays the product name and price, while the DiscountedProduct class will also display the discount and the final price.

- **Implement a Default ReportProduct Method in ProductBase:** Implement a default ReportProduct method in the ProductBase class to print the basic product details.
- **Override the ReportProduct Method in the DiscountedProduct Class:** Override the ReportProduct method in the DiscountedProduct class to display the discount percentage and the final discounted price.

Challenge Tasks (Optional):

- **Support Tiered Discounts in DiscountedProduct:** Modify the DiscountedProduct class to support tiered discounts. For example, if the price is above £100, a 10%

discount is applied, and if the price is above £200, a 20% discount is applied. How would you adjust the CalculateFinalPrice() method to support this logic?

- **Include Validation Logic in ProductBase:** Modify the ProductBase class to include validation logic before creating a product. For example, check if the price is greater than zero or if the product name is not null or empty. Throw appropriate exceptions if validation fails. How would you implement this validation, and what exceptions would you throw?