Task 1: Implement a Bank Account System

Objective:
To practice the use of **Encapsulation** and **Methods** to simulate a bank account with basic operations like deposit, withdrawal, and balance checking.

Steps:

1. Define a `BankAccount` class with the following attributes:
    - `account_holder`: (string) the name of the account holder.
    - `balance`: (float) the account balance (initially set to 0).
2. Add the following methods to the `BankAccount` class:
    - `deposit(amount)`: Adds the amount to the balance.
    - `withdraw(amount)`: Subtracts the amount from the balance, ensuring that the withdrawal amount does not exceed the balance.
    - `get_balance()`: Returns the current balance.
3. Make the `balance` attribute private (use double underscores) to ensure it cannot be directly modified outside the class.
4. Create an instance of the `BankAccount` class for a customer and perform several transactions (deposit and withdraw).

Question 1:

- What will happen if you try to access the `balance` attribute directly outside the class? How does encapsulation help in this scenario?

Question 2:

- How would you modify the `deposit` and `withdraw` methods to raise an exception if a user tries to withdraw more than their available balance?

Task 2: Inheritance with Animals

Objective:
To understand **Inheritance** by creating different animal types that share common properties.

Steps:

1. Create a parent class called `Animal` with:
    - `name`: (string) the name of the animal.
    - `age`: (integer) the age of the animal.
    - `eat()`: A method that prints a message, e.g., "The animal is eating."
2. Create two child classes:
    - `Dog`: Inherits from `Animal`. Adds a new method `bark()`, which prints "Woof!"
    - `Cat`: Inherits from `Animal`. Adds a new method `meow()`, which prints "Meow!"
3. Instantiate objects of `Dog` and `Cat` and call methods from both the parent and child classes.

Question 1:

- What is the advantage of using inheritance in this scenario? How does it reduce redundancy?

Question 2:

- What would happen if we try to call the `eat()` method on a `Cat` object? Why?

Task 3: Polymorphism with Shape Classes

Objective:
To explore **Polymorphism** by defining a base class and creating subclasses that implement specific behaviors.

Steps:

1. Create a base class `Shape` with:
    - `name`: (string) the name of the shape.
    - An abstract method `area()` to calculate the area (use `abc` module).
2. Create two child classes:
    - `Circle`: Implements the `area()` method to calculate the area of a circle (π * r^2).
    - `Rectangle`: Implements the `area()` method to calculate the area of a rectangle (width * height).
3. Create a function `print_area(shape)` that takes any `Shape` object and prints the area by calling the `area()` method.
4. Instantiate both `Circle` and `Rectangle` objects, and call the `print_area()` function with both objects.

Question 1:

- How does polymorphism allow the `print_area()` function to handle objects of different types (`Circle` and `Rectangle`)?

Question 2:

- What would happen if the `Rectangle` class did not implement the `area()` method? How would this affect the `print_area()` function?

Task 4: Abstract Class for Vehicle

Objective:
To learn **Abstraction** by creating a common structure for different vehicle types using an abstract class.

Steps:

1. Define an abstract class `Vehicle` with the following methods:
    - `start()`: Starts the vehicle.
    - `stop()`: Stops the vehicle.
    - `drive()`: Drives the vehicle.

    All methods should raise a `NotImplementedError`, as they will be implemented by subclasses.

2. Create two subclasses:
    - `Car`: Implements the `start()`, `stop()`, and `drive()` methods.
    - `Bike`: Implements the `start()`, `stop()`, and `drive()` methods.
3. Create objects of `Car` and `Bike` and invoke all three methods (`start()`, `stop()`, and `drive()`) on both objects.

Question 1:

- Why is the `Vehicle` class abstract? What is the purpose of the `NotImplementedError` in this case?

Question 2:

- How would the program behave if the `Car` class did not implement the `start()` method?

Task 5: Class Composition with a Shopping Cart

Objective:
To understand **Composition** in OOP by creating a system where objects are made up of other objects.

Steps:

1. Create a `Product` class with:
    - `name`: (string) the name of the product.
    - `price`: (float) the price of the product.
2. Create a `ShoppingCart` class that holds a list of `Product` objects. The class should have the following methods:
    - `add_product(product)`: Adds a product to the cart.
    - `remove_product(product_name)`: Removes a product by name.
    - `total_price()`: Returns the total price of all the products in the cart.
3. Instantiate several `Product` objects and add them to a `ShoppingCart`.

Question 1:

- What is the relationship between `ShoppingCart` and `Product` in this task? How does composition differ from inheritance?

Question 2:

- How would you modify the `ShoppingCart` to handle the scenario where the same product is added multiple times?

Task 6: Design a Library System

Objective:
To practice **OOP concepts** by designing a library system that allows adding and borrowing books.

Steps:

1. Create a `Book` class with:
    - `title`: (string) the title of the book.
    - `author`: (string) the author of the book.
    - `is_borrowed`: (boolean) indicates if the book is borrowed or not.
    - `borrow()`: Method to mark the book as borrowed.
    - `return_book()`: Method to mark the book as returned.
2. Create a `Library` class with:
    - `books`: A list that holds all the books.
    - `add_book(book)`: Adds a book to the library.
    - `borrow_book(title)`: Allows borrowing a book by title, and updates the `is_borrowed` status.
    - `return_book(title)`: Allows returning a book and updates the `is_borrowed` status.
3. Create several `Book` objects and add them to a `Library`. Simulate borrowing and returning books.

Question 1:

- How does the `Library` class manage the relationship between the library and its books? How does encapsulation help here?

Question 2:

- If a user tries to borrow a book that is already borrowed, how would you handle that in your code? Would you raise an exception or print a message?

Task 7: Creating an E-commerce System

Objective:
To practice **Encapsulation**, **Inheritance**, and **Polymorphism** by creating a simple e-commerce system with different product types.

Steps:

1. Create a base class `Product` with:
    - `name`: (string) the product name.
    - `price`: (float) the product price.
    - `get_info()`: A method to return the product name and price.
2. Create two subclasses:
    - `Electronics`: Inherits from `Product`. Adds a `warranty_period` attribute and overrides the `get_info()` method to include warranty information.
    - `Clothing`: Inherits from `Product`. Adds a `size` attribute and overrides the `get_info()` method to include size information.
3. Create a `ShoppingCart` class that holds a list of `Product` objects, with methods to:
    - Add products to the cart.
    - Calculate the total price of all products in the cart.
4. Test the system by adding `Electronics` and `Clothing` products to the cart and printing out the product details.

Question 1:

- How does inheritance help in reducing the redundancy of code between `Electronics` and `Clothing`?

Question 2:

- How does polymorphism work when you call `get_info()` on different product types (e.g., `Electronics` and `Clothing`)?