Lab_10: OOP principles.

Variants:

1. **Encapsulation:**
    - Create a class representing a bank account. Use encapsulation to protect the account balance and provide methods for deposit and withdrawal.
2. **Inheritance:**
    - Implement a base class `Shape` with methods for calculating area and perimeter. Create subclasses like `Circle` and `Rectangle` that inherit from `Shape` and override necessary methods.
3. **Polymorphism:**
    - Define a class `Animal` with a method `make_sound()`. Create subclasses like `Dog` and `Cat` that implement their own version of `make_sound()`.
4. **Abstraction:**
    - Design a class `Vehicle` with attributes such as speed and fuel efficiency. Use abstraction to hide the internal details and provide methods for acceleration and fuel consumption.
5. **Composition:**
    - Create a class `Car` that has a composition relationship with a class `Engine`. Demonstrate how the `Car` class can utilize the functionalities of the `Engine` class.
6. **Interfaces:**
    - Define an interface `Drawable` with a method `draw()`. Implement classes like `Circle` and `Square` that implement the `Drawable` interface.
7. **Dependency Injection:**
    - Design a class `Logger` for logging messages. Create another class `Calculator` that depends on the `Logger` class for logging operations.
8. **Encapsulation and Access Control:**
    - Create a class representing a university. Use encapsulation and access control to manage data such as student information and grades.
9. **Inheritance and Method Overriding:**
    - Design a class `Person` with attributes like name and age. Create a subclass `Employee` that inherits from `Person` and overrides the `display_info()` method.
10. **Abstract Classes:**
    - Create an abstract class `Shape` with abstract methods for calculating area and perimeter. Implement concrete subclasses like `Circle` and `Rectangle`.
11. **Polymorphic Relationships:**
    - Design classes representing different types of employees (e.g., full-time, part-time). Demonstrate a polymorphic relationship by creating a list of employees with different types.
12. **Composition vs. Inheritance:**

- Compare and contrast the use of composition and inheritance by implementing a scenario (e.g., modeling a zoo with animals).

13. **Observer Pattern:**
    - Implement the observer pattern by creating a class `Subject` with a list of observers. When the state of the `Subject` changes, notify all registered observers.
14. **Factory Method:**
    - Create a class `Document` with a factory method for creating instances of different document types (e.g., `PDFDocument`, `WordDocument`).
15. **Strategy Pattern:**
    - Implement the strategy pattern by creating a class `PaymentProcessor` with different payment strategies (e.g., credit card, PayPal) that can be switched at runtime.
16. **Template Method Pattern:**
    - Design a class `Report` with a template method for generating reports. Allow subclasses to override specific steps in the report generation process.
17. **Decorator Pattern:**
    - Implement the decorator pattern by creating a class `Coffee` and decorators for adding extras like sugar, milk, and flavor.
18. **Chain of Responsibility:**
    - Design a chain of handler classes to process requests in a sequential manner, passing the request to the next handler in the chain.
19. **Singleton Pattern:**
    - Implement a singleton pattern for a logging class to ensure that only one instance of the logger is created throughout the program.
20. **Command Pattern:**
    - Create a class `Command` with an execute method. Implement concrete command classes and a class `Invoker` that can execute different commands.