OOPS Programs

Class and Object Basics:

1.Create a class "Person" with attributes name and age. Add a method to display the details.
2.Create an object of the "Person" class and display its details.
3.Implement a class "Car" with attributes make, model, and year. Display the car details.
4.Create an object of the "Car" class and display its details.

Encapsulation:

1.Implement a class "BankAccount" with private attributes accountNumber and balance. Add
2.methods to deposit and withdraw money.
3.Create an object of "BankAccount" and perform deposit and withdrawal operations.

Inheritance:

1.Create a base class "Shape" with methods to calculate area and perimeter.
2.Derive classes "Rectangle" and "Circle" from the "Shape" class and implement area and perimeter calculations.

Polymorphism:

1.Define an interface "Shape" with a method "calculateArea."
2.Implement classes "Circle" and "Rectangle" that implement the "Shape" interface and calculate their areas.

Abstraction:

1.Create an abstract class "Animal" with abstract methods "eat" and "sound."
2.Implement classes "Dog" and "Cat" that extend the "Animal" class and implement the abstract methods.

Constructor Overloading:

1.Modify the "Person" class to have multiple constructors with different parameters.
2.Create objects of the "Person" class using different constructors.

Method Overloading:

1.Add a method "add" to the "Calculator" class that can take two integers or three integers and perform addition accordingly.
2.Use the "add" method with both two and three integers.

Operator Overloading:

1.Overload the "+" operator for a class representing a complex number.
2.Add two complex numbers using the overloaded "+" operator.

Static Methods:

1.Implement a class "MathOperations" with a static method to find the square root of a number.
2.Use the static method without creating an object of the class.

Singleton Pattern:

1.Create a singleton class "Logger" that logs messages.
2.Use the singleton instance to log messages from different parts of the program.

Composition:

1.Create classes "Engine" and "Car" where "Car" has an "Engine" object.
2.Demonstrate the composition relationship between "Car" and "Engine."

Interface:

1.Define an interface "Drawable" with a method "draw."
2.Implement classes "Circle" and "Rectangle" that implement the "Drawable" interface.

Abstract Factory:

1.Create an abstract class "AbstractFactory" with methods to create "ProductA" and "ProductB."
2.Implement concrete classes "ConcreteFactory1" and "ConcreteFactory2" that extend the abstract factory.

Observer Pattern:

1.Implement an observer pattern where multiple objects observe changes in a subject.
2.Notify the observers when a change occurs in the subject.

Decorator Pattern:

1.Create a class "Coffee" with a method "cost."
2.Implement decorators like "Milk" and "Sugar" to modify the cost of a "Coffee" object.

Strategy Pattern:

1.Define an interface "PaymentStrategy" with methods to perform payment.
2.Implement classes like "CreditCardPayment" and "PayPalPayment" that implement the "PaymentStrategy."

Command Pattern:

1.Create a class "Command" with an execute method.
2.Implement concrete commands like "LightOnCommand" and "LightOffCommand" for a home automation system.

Factory Method:

1.Define an interface "Document" with a method "print."
2.Implement classes like "PDFDocument" and "WordDocument" that implement the "Document" interface.

Iterator Pattern:

1.Create an interface "Iterator" with methods to iterate over a collection.
2.Implement an iterator for a custom collection.

Memento Pattern:

1.Implement a class "Originator" that has a state.
2.Create a "Memento" class to store the state and a "Caretaker" class to manage multiple states.

State Pattern:

1.Define a "Context" class that can change its state.
2.Implement different states like "StateA" and "StateB" that the "Context" class can transition between.

Proxy Pattern:

1.Create an interface "Image" with a method "display."
2.Implement a "RealImage" class that implements the "Image" interface and a "ProxyImage" class that controls access to the "RealImage."

Chain of Responsibility:

1.Implement a chain of handlers to process requests.
2.Each handler should decide whether to handle the request or pass it to the next handler.

Template Method Pattern:

1.Create an abstract class "Game" with template methods "initialize," "startPlay," and "endPlay."
2.Implement classes like "Football" and "Basketball" that extend the "Game" class.

Prototype Pattern:

1.Implement a prototype pattern for creating copies of objects.
2.Clone objects using the prototype pattern.

Bridge Pattern:

1.Define an interface "Color" with a method "fill."
2.Implement concrete classes like "Red" and "Blue" that implement the "Color" interface.
3.Create an abstract class "Shape" that uses the "Color" interface and implement concrete shapes.

Composite Pattern:

1.Create an interface "Component" with a method "display."

2.Implement classes like "Leaf" and "Composite" that implement the "Component" interface.

Adapter Pattern:

1.Create an interface "Target" with a method "request."
2.Implement classes like "Adaptee" that have a method incompatible with "Target."
3.Create an adapter class to make "Adaptee" compatible with "Target."

Mediator Pattern:

1.Define a "Mediator" interface with methods for communication between components.
2.Implement classes like "Colleague" that communicate through the mediator.

Visitor Pattern:

1.Create an interface "Visitor" with methods to visit different elements.
2.Implement classes like "ElementA" and "ElementB" that accept visitors.

Composite Pattern:

1.Define a "Component" interface with methods to add and remove components.
2.Implement classes like "Leaf" and "Composite" that implement the "Component" interface.

Command Pattern:

1.Create a remote control that can execute different commands.
2.Implement commands like "TurnOnCommand" and "TurnOffCommand" for various devices.

Observer Pattern:

1.Implement a stock market system where different observers receive updates on stock prices.
2.Notify observers when stock prices change.

Proxy Pattern:

1.Create a proxy for a costly resource, allowing access only if certain conditions are met.
2.Use the proxy to control access to the resource.

Factory Method:

1.Implement a factory method to create different types of vehicles.
2.Create instances of vehicles using the factory method.

State Pattern:

1.Create a context class representing a traffic light that changes its state.
2.Implement states like "Red," "Yellow," and "Green."

Abstract Factory:

1.Define an abstract factory for creating furniture.
2.Implement concrete factories for modern and Victorian furniture.

Chain of Responsibility:

1.Create a chain of loggers to process log messages.
2.Each logger decides whether to handle the message or pass it to the next logger.

Decorator Pattern:

1.Implement a text editor with decorators like "Bold" and "Italic" to modify the text.
2.Apply multiple decorators to the text.

Bridge Pattern:

1.Define a bridge between different drawing shapes and rendering engines.
2.Implement concrete shapes and rendering engines.

Composite Pattern:

1.Create a file system representation using the composite pattern.
2.Implement classes like "File" and "Directory."

Strategy Pattern:

1.Implement a billing system that uses different billing strategies based on customer type.
2.Define strategies like "NormalBilling" and "DiscountBilling."

Visitor Pattern:

1.Create a document structure and implement a visitor to perform operations on the document elements.
2.Define elements like "Paragraph" and "Image."

State Pattern:

1.Implement a vending machine with different states like "NoCoinState" and "HasCoinState."
2.Change the vending machine state based on user actions.

Abstract Factory:

1.Define an abstract factory for creating UI components.
2.Implement concrete factories for desktop and mobile UIs.

Observer Pattern:

1.Implement a weather station where different observers receive updates on temperature and humidity.
2.Notify observers when weather conditions change.

Command Pattern:

1.Create a remote control for a smart home with different commands for controlling devices.
2.Implement commands like "TurnOnLightsCommand" and "TurnOffACCommand."

Prototype Pattern:

1.Implement a prototype pattern for creating different types of animals.
2.Clone animal objects using the prototype pattern.
3.These exercises cover a wide range of OOP concepts and can be adapted to various programming languages. Feel free to choose the ones that align with your language of choice and programming environment.