# 20 OOPs Practical Coding Questions

This document contains 20 critical Object-Oriented Programming (OOP) coding questions, designed for students to enhance their understanding of core OOP concepts like Encapsulation, Abstraction, Polymorphism, Inheritance, Interfaces, and Design Patterns. The total duration is 2 hours (120 minutes).

|  |  |  |
| --- | --- | --- |
| No. | Question | Duration (Minutes) |
| 1 | \*\*Encapsulation Challenge\*\*  Create a `BankAccount` class where `balance` is a private field. Implement methods for `Deposit()` and `Withdraw()`, ensuring that withdrawal is only allowed if there is a sufficient balance. Prevent direct modification of `balance` from outside the class. | 6 |
| 2 | \*\*Data Hiding with Properties\*\*  Create a `Student` class where `Name` and `RollNo` are private fields. Use properties to enforce validation (e.g., `RollNo` cannot be negative, `Name` cannot be empty). | 6 |
| 3 | \*\*Constructor Overloading in a Library System\*\*  Implement a `Book` class with three different constructors: (1) Default constructor, (2) Constructor with `Title` and `Author`, (3) Constructor with `Title`, `Author`, and `ISBN`. Ensure each constructor initializes properties correctly. | 6 |
| 4 | \*\*Abstraction with Abstract Classes\*\*  Design an abstract class `Shape` with an abstract method `CalculateArea()`. Create derived classes `Circle` and `Rectangle` that implement this method. Demonstrate abstraction by instantiating these classes. | 7 |
| 5 | \*\*Method Overriding for a Vehicle System\*\*  Create a base class `Vehicle` with a method `Start()`. Override it in `Car` and `Bike` classes to provide specific implementations. Use the `override` keyword and demonstrate polymorphism. | 7 |
| 6 | \*\*Using Virtual Methods in Inheritance\*\*  Create a `Person` base class with a `GetDetails()` method. Derive `Student` and `Teacher` classes that override `GetDetails()` to display their respective properties. Call `GetDetails()` using a base class reference. | 7 |
| 7 | \*\*Method Overloading in a Calculator Class\*\*  Implement a `Calculator` class with overloaded methods `Add()`. It should accept two integers, three integers, and two double values separately. Demonstrate how method overloading works. | 6 |
| 8 | \*\*Interface Implementation Challenge\*\*  Define an interface `IPlayable` with a method `Play()`. Implement this interface in `MusicPlayer` and `VideoPlayer` classes with different implementations. | 7 |
| 9 | \*\*Multiple Inheritance Using Interfaces\*\*  Define two interfaces `IPrintable` (for printing details) and `ISerializable` (for saving to a file). Implement both in a `Report` class and demonstrate multiple interface implementation. | 7 |
| 10 | \*\*Design a Role-Based Access System\*\*  Create a base class `User` with properties like `Username` and `Role`. Derive `Admin` and `Customer` classes that override a method `AccessControl()`, where `Admin` can access all features, but `Customer` has limited access. | 7 |
| 11 | \*\*Operator Overloading for Complex Numbers\*\*  Create a `ComplexNumber` class with properties `Real` and `Imaginary`. Overload the `+` operator to add two complex numbers. | 8 |
| 12 | \*\*Shallow Copy vs. Deep Copy\*\*  Create a `Department` class with a reference-type property `Manager`. Implement both \*\*Shallow Copy\*\* and \*\*Deep Copy\*\* to show how references are handled. | 7 |
| 13 | \*\*Static Members in a Banking System\*\*  Implement a `Bank` class with a static field `InterestRate` and a static method `SetInterestRate()`. Show how static members work across multiple objects. | 6 |
| 14 | \*\*Sealed Class in a Security System\*\*  Create a sealed class `SecuritySystem` that prevents inheritance. Show how sealing a class stops further extension. | 5 |
| 15 | \*\*Use of Delegates for Event Handling\*\*  Create a `Button` class that has a delegate `OnClick`. Implement an event that triggers when the button is clicked. | 8 |
| 16 | \*\*Factory Pattern for Object Creation\*\*  Implement a `VehicleFactory` class that returns different vehicle objects (`Car`, `Bike`) based on an input parameter. | 8 |
| 17 | \*\*Decorator Pattern in a Logger System\*\*  Implement an `ILogger` interface and `FileLogger` class. Use the \*\*Decorator Pattern\*\* to add extra logging features like timestamp and error categorization. | 8 |
| 18 | \*\*Thread-Safe Singleton Pattern for Configuration Management\*\*  Implement a Singleton class `ConfigurationManager` that prevents multiple instances. Ensure it is thread-safe. | 7 |
| 19 | \*\*Real-World Observer Pattern for Notifications\*\*  Implement an `INotificationObserver` interface where `EmailNotifier` and `SMSNotifier` listen for notifications. When a new message arrives, all observers should receive an update. | 7 |
| 20 | \*\*Strategy Pattern for Dynamic Discount Calculation\*\*  Implement different discount strategies (`NoDiscount`, `PercentageDiscount`, `FixedAmountDiscount`). The `ShoppingCart` class should apply a strategy dynamically based on user input. | 7 |