Contents

1. What is Object-Oriented Programming (OOP)?

- **Definition:** OOP is a programming paradigm based on the concept of "objects," which can contain data and code. Objects represent real-world entities and can encapsulate properties (data) and behaviors (methods).
- **Key Concepts:** Encapsulation, inheritance, polymorphism, and abstraction.
- **Advantages:** Promotes code reusability, scalability, and maintainability.

Example:

csharp
Copy code
```
public class Car {
    public string Color { get; set; }
    public string Model { get; set; }

    public void Drive() {
        Console.WriteLine($"The {Color} {Model} is driving.");
    }
}

Car myCar = new Car { Color = "Red", Model = "Toyota" };
myCar.Drive(); // Output: The Red Toyota is driving.
```

---

2. What is the difference between procedural and Object-Oriented programming?

- **Procedural Programming:**
    - Focuses on functions or procedures to operate on data.
    - Code is organized in a linear manner.
    - Example: C, Pascal.
- **Object-Oriented Programming:**
    - Centers around objects that combine data and behavior.
    - Supports abstraction, encapsulation, inheritance, and polymorphism.
    - Example: C#, Java.

Example of Procedural vs. OOP:

- **Procedural:**

csharp
Copy code

```
int Add(int a, int b) {
    return a + b;
}
int result = Add(3, 4); // Result is 7
```

- **OOP:**

```
csharp
Copy code
public class Calculator {
    public int Add(int a, int b) {
        return a + b;
    }
}

Calculator calc = new Calculator();
int result = calc.Add(3, 4); // Result is 7
```

---

3. What is encapsulation?

- **Definition:** Encapsulation is the bundling of data and methods that operate on that data within a single unit (class). It restricts direct access to some components, which is a means of preventing unintended interference and misuse of the methods and data.
- **Implementation:** Achieved using access specifiers (public, private, protected).
- **Benefits:** Enhances security and reduces complexity.

Example:

```
csharp
Copy code
public class BankAccount {
    private double balance;
    public void Deposit(double amount) {
        if (amount > 0) {
            balance += amount;
        }
    }

    public double GetBalance() {
        return balance;
    }
}

BankAccount account = new BankAccount();
account.Deposit(100);
Console.WriteLine(account.GetBalance()); // Output: 100
```

---

4. What is polymorphism? Explain overriding and overloading.

- **Polymorphism:** The ability of different classes to be treated as instances of the same class through a common interface. It allows for method overriding and overloading.
- **Overriding:** Redefining a base class method in a derived class.
- **Overloading:** Defining multiple methods with the same name but different signatures (parameters).

Example of Overriding:

csharp
Copy code

```
public class Animal {
    public virtual void Speak() {
        Console.WriteLine("Animal speaks");
    }
}

public class Dog : Animal {
    public override void Speak() {
        Console.WriteLine("Bark");
    }
}

Animal myDog = new Dog();
myDog.Speak(); // Output: Bark
```

Example of Overloading:

csharp
Copy code
```
public class MathOperations {
    public int Add(int a, int b) {
        return a + b;
    }

    public double Add(double a, double b) {
        return a + b;
    }
}

MathOperations math = new MathOperations();
Console.WriteLine(math.Add(2, 3)); // Output: 5
Console.WriteLine(math.Add(2.5, 3.5)); // Output: 6
```

---

5. What is inheritance? Name some types of inheritance.

- **Definition:** Inheritance is a mechanism where one class (derived class) can inherit properties and methods from another class (base class).
- **Benefits:** Promotes code reusability and establishes a relationship between classes.
- **Types of Inheritance:**
    - **Single Inheritance:** A class inherits from one base class.

    - **Multiple Inheritance:** A class inherits from multiple classes (not supported in C#).
    - **Multilevel Inheritance:** A class inherits from a derived class.
    - **Hierarchical Inheritance:** Multiple classes inherit from a single base class.

Example of Inheritance:

csharp
Copy code
```
public class Vehicle {
   public void Start() {
      Console.WriteLine("Vehicle started");
   }
}

public class Car : Vehicle {
   public void Drive() {
      Console.WriteLine("Car is driving");
   }
}

Car myCar = new Car();
myCar.Start(); // Output: Vehicle started
myCar.Drive(); // Output: Car is driving
```

---

6. What is an abstraction? Name some abstraction techniques.

- **Definition:** Abstraction is the concept of hiding the complex implementation details and exposing only the necessary features of an object.
- **Techniques:**
    - **Abstract Classes:** Cannot be instantiated and may contain abstract methods.

- **Interfaces:** Define contracts for classes without implementing methods.

Example of Abstraction with Abstract Class:

```
csharp
Copy code
public abstract class Shape {
    public abstract double Area();
}

public class Circle : Shape {
    private double radius;

    public Circle(double radius) {
        this.radius = radius;
    }

    public override double Area() {
        return Math.PI * radius * radius;
    }
}

Shape myCircle = new Circle(5);
Console.WriteLine(myCircle.Area()); // Output: 78.53981633974483
```

Example of Abstraction with Interface:

```
csharp
Copy code
public interface IShape {
    double Area();
}

public class Rectangle : IShape {
    private double width;
    private double height;
    public Rectangle(double width, double height) {
        this.width = width;
        this.height = height;
    }

    public double Area() {
        return width * height;
    }
}

IShape myRectangle = new Rectangle(4, 5);
Console.WriteLine(myRectangle.Area()); // Output: 20
```

---

7. What is a class in OOP?

- **Definition:** A class is a blueprint for creating objects, defining properties and methods that those objects will have.
- **Components:**
  - **Fields:** Variables to hold data.
  - **Methods:** Functions to define behavior.
  - **Constructors:** Special methods for initializing objects.

Example:

csharp
Copy code
```
public class Person {
    public string Name { get; set; }
    public int Age { get; set; }

    public Person(string name, int age) {
        Name = name;
        Age = age;
    }
    public void Introduce() {
        Console.WriteLine($"Hello, my name is {Name} and I am {Age} years old.");
    }
}

Person person = new Person("Alice", 30);
person.Introduce(); // Output: Hello, my name is Alice and I am 30 years old.
```

---

8. What is an object in OOP?

- **Definition:** An object is an instance of a class that contains data and methods defined by that class.
- **Characteristics:** Objects can interact with one another and can be manipulated using methods.

Example:

```
csharp
Copy code
Person person1 = new Person("Bob", 25);
Person person2 = new Person("Alice", 30);

person1.Introduce(); // Output: Hello, my name is Bob and I am 25 years old.
person2.Introduce(); // Output: Hello, my name is Alice and I am 30 years old.
```

---

9. How do access specifiers work and what are they typically?

- **Definition:** Access specifiers define the visibility of classes, methods, and other members.
- **Common Specifiers:**
    - **public:** Accessible from anywhere.
    - **private:** Accessible only within the class.

- protected: Accessible within the class and by derived class instances.
- **internal:** Accessible only within the same assembly.

Example:

```
csharp
Copy code
public class Example {
    private int privateField;
    protected int protectedField;
    public int publicField;
    internal int internalField;

    public void Display() {
        Console.WriteLine($"Public: {publicField}, Protected: {protectedField}, Internal: {internalField}");
    }
}
```

---

10. Name some ways to overload a method.

- **Parameter Types:** Different data types for parameters.
- **Number of Parameters:** Different number of parameters.
- **Order of Parameters:** Different order of parameter types.

Example:

```
csharp
Copy code
public class OverloadExample {
    public int Add(int a, int b) {
        return a + b;
    }

    public double Add(double a, double b) {
        return a + b;
    }

    public int Add(int a, int b, int c) {
        return a + b + c;
    }
}

OverloadExample example = new OverloadExample();
Console.WriteLine(example.Add(3, 4)); // Output: 7
Console.WriteLine(example.Add(3.5, 4.5)); // Output: 8
Console.WriteLine(example.Add(1, 2, 3)); // Output: 6
```

---

11. What is cohesion in OOP?

- **Definition:** Cohesion refers to how closely related and focused the responsibilities of a single module (class) are.
- **High Cohesion:** Means that a class has a clear purpose and its methods and properties are closely related.
- **Benefits:** Makes the class easier to understand, maintain, and reuse.

Example of High Cohesion:

csharp
Copy code
```
public class Order {
    public void AddItem(Item item) { /* Implementation */ }
    public void RemoveItem(Item item) { /* Implementation */ }
    public void CalculateTotal() { /* Implementation */ }
}
```

---

12. What is coupling in OOP?

- **Definition:** Coupling refers to the degree of direct knowledge that one class has about another class.
- **Loose Coupling:** Indicates that classes are independent and interact through interfaces or abstract classes, promoting flexibility and reusability.
- **Tight Coupling:** Indicates that classes are heavily dependent on one another, making changes difficult.

Example of Loose Coupling:

csharp

Copy code

```
public interface INotification {
  void Notify(string message);
}

public class EmailNotification : INotification {
  public void Notify(string message) {
    Console.WriteLine($"Email: {message}");
  }
}

public class User {
  private INotification notification;

  public User(INotification notification) {
    this.notification = notification;
  }

  public void Register() {
    notification.Notify("User registered");
  }
}

INotification emailNotifier = new EmailNotification();
User user = new User(emailNotifier);
user.Register(); // Output: Email: User registered
```

13. What is a constructor and how is it used?

- **Definition:** A constructor is a special method invoked when an object is created. It initializes the object’ s properties.
- **Characteristics:**
    - Has the same name as the class.
    - Does not have a return type.

Example:

csharp

Copy code

```
public class Book {
  public string Title { get; set; }
  public string Author { get; set; }

  // Constructor
  public Book(string title, string author) {
    Title = title;
    Author = author;
  }
}

Book myBook = new Book("1984", "George Orwell");
Console.WriteLine($"{myBook.Title} by {myBook.Author}"); // Output: 1984 by George Orwell
```

14. Describe the concept of destructor or finalizer in OOP.

- **Definition:** A destructor is a special method invoked when an object is being destroyed or garbage collected. It is used to free up resources held by the object.
- **Syntax:** Defined using the ~ symbol before the class name.

- **Note:** Destructors are not commonly used in C# due to garbage collection.

Example:

```
csharp
Copy code
public class Resource {
   // Destructor
   ~Resource() {
      Console.WriteLine("Destructor called, resource freed.");
   }
}

Resource res = new Resource();
// Destructor will be called when the object is garbage collected
```

---

15. Compare inheritance vs. mixin vs. composition.

- **Inheritance:**
  - Inherits behavior and properties from a base class.
  - Represents an "is-a" relationship.
- **Mixin:**
  - A way to include methods and properties from multiple sources.
  - Not natively supported in C#, often simulated through interfaces.
- **Composition:**
  - Combines objects to create more complex types.
  - Represents a "has-a" relationship.

Example of Composition:

```
csharp
Copy code
public class Engine {
   public void Start() {
        Console.WriteLine("Engine started");
    }
}

public class Car {
    private Engine engine = new Engine();

    public void Start() {
        engine.Start(); // Delegation
        Console.WriteLine("Car started");
    }
}

Car myCar = new Car();
myCar.Start(); // Output: Engine started
               //         Car started
```

---

16. Explain the concept of an interface and how it differs from an abstract class.

- **Interface:**
  - Defines a contract for classes without implementing any methods.
  - Can be implemented by multiple classes.
  - Does not support access modifiers.
- **Abstract Class:**
  - Can contain both implemented and unimplemented methods.
  - Can maintain state (fields).
  - Can have access modifiers.

Example of Interface:

csharp
Copy code
```
public interface IDrawable {
    void Draw();
}

public class Circle : IDrawable {
    public void Draw() {
        Console.WriteLine("Drawing a circle");
    }
}
```

Example of Abstract Class:

```
csharp
Copy code
public abstract class Shape {
    public abstract double Area(); // Abstract method
    public void Display() {
        Console.WriteLine("Shape information");
    }
}

public class Rectangle : Shape {
    public double Width { get; set; }
    public double Height { get; set; }

    public override double Area() {
        return Width * Height;
    }
}
```

---

17. Can a class have multiple parents in a single-inheritance system?

- **Answer:** No, C# does not support multiple inheritance directly. A class can inherit from one base class only. However, a class can implement multiple interfaces.

Example:

csharp
Copy code
```
public interface IFlyable {
    void Fly();
}

public interface ISwimmable {
    void Swim();
}

public class Duck : IFlyable, ISwimmable {
    public void Fly() {
        Console.WriteLine("Duck is flying");
    }

    public void Swim() {
        Console.WriteLine("Duck is swimming");
    }
}
```

---

18. How would you design a class to prevent it from being subclassed?

- **Answer:** You can declare the class as `sealed`, which prevents any class from inheriting from it.

Example:

csharp
Copy code
```
public sealed class FinalClass {
    public void Show() {
        Console.WriteLine("This is a sealed class.");
    }
}
```

```
// This will cause a compile-time error
// public class DerivedClass : FinalClass { }
```

---

19. Explain the 'is-a' vs 'has-a' relationship in OOP.

- **Is-a Relationship:**
  - Represents inheritance. If a class A is derived from class B, A "is-a" B.
  - Example: A Dog is a type of Animal.

Example:

csharp
Copy code
```
public class Animal { }
public class Dog : Animal { } // Dog is-a Animal
```

- **Has-a Relationship:**
  - Represents composition. If class A contains an object of class B, A "has-a" B.
  - Example: A Car has an Engine.

Example:

csharp
Copy code
```
public class Engine { }
public class Car {
   private Engine engine = new Engine(); // Car has-a Engine
}
```

---

20. Explain how aggregation relationship is represented in OOP.

- **Aggregation:** A special type of association that represents a "whole-part" relationship, where the part can exist independently of the whole.
- **Implementation:** Achieved by including references to the part class in the whole class.

Example:

```
csharp
Copy code
public class Department {
   public string Name { get; set; }
}

public class Company {
   private List<Department> departments = new List<Department>();

   public void AddDepartment(Department department) {
      departments.Add(department);
   }
}

// A department can exist independently of the company
Department sales = new Department { Name = "Sales" };
Company company = new Company();
company.AddDepartment(sales);
```

21. What is method overriding, and what rules apply to it?

- **Definition:** Method overriding allows a derived class to provide a specific implementation of a method that is already defined in its base class.
- **Rules:**
  - The method in the derived class must have the same name, return type, and parameters as the method in the base class.

- The base method must be marked with the virtual keyword, and the overriding method in the derived class must use the override keyword.
- The method must not be more restrictive in terms of access modifiers.

Example:

csharp
Copy code
```
public class Animal {
   public virtual void Speak() {
      Console.WriteLine("Animal speaks");
   }
}

public class Dog : Animal {
   public override void Speak() {
      Console.WriteLine("Bark");
   }
}

Animal myDog = new Dog();
myDog.Speak(); // Output: Bark
```

---

22. Describe the use of static methods and when they are appropriate.

- **Definition:** Static methods belong to the class itself rather than to any specific object instance. They can be called without creating an object of the class.
- **When to Use:**
  - When a method does not need access to instance variables or instance methods.
  - For utility or helper methods that perform operations that are independent of object state.
  - When a method needs to maintain shared state across all instances.

Example:

```
csharp
Copy code
public class MathHelper {
    public static int Add(int a, int b) {
        return a + b;
    }
}

int result = MathHelper.Add(3, 4); // Output: 7
```

---

23. What is multiple inheritance, and what are some of its disadvantages?

- **Definition:** Multiple inheritance occurs when a class can inherit from more than one base class.
- **Disadvantages:**
    - **Complexity:** Increases complexity in the class hierarchy and can make code harder to understand.
    - **Ambiguity:** Can lead to ambiguity when methods or properties from multiple base classes have the same name.
    - **Diamond Problem:** A specific issue where a class inherits from two classes that both inherit from a common base class.

Example (not C# code, as C# does not support multiple inheritance):

```
plaintext
Copy code
class A { }
class B : A { }
class C : A { }
class D : B, C { } // This will cause ambiguity issues
```

---

24. Can you explain the 'diamond problem' in multiple inheritance?

- **Definition:** The diamond problem occurs when a class inherits from two classes that both inherit from a common base class. This can create ambiguity for methods or properties inherited from the base class.
- **Illustration:**

plaintext
Copy code
```
   A
  / \
 B   C
  \ /
   D
```

- **In C#:** C# does not allow multiple inheritance of classes, so this problem is avoided. However, it can occur with interfaces.

Example with interfaces:

csharp
Copy code
```
public interface IA {
    void DoSomething();
}

public interface IB : IA {
    void DoSomethingElse();
}

public interface IC : IA {
    void DoAnotherThing();
}

public class D : IB, IC {
    public void DoSomething() {
        // Implementation
    }

    public void DoSomethingElse() {
        // Implementation
    }

    public void DoAnotherThing() {
        // Implementation
    }
}
```

---

25. How do OOP languages support polymorphism under the hood?

- **Mechanism:** OOP languages implement polymorphism through dynamic dispatch. This means that the method to be called is determined at runtime based on the object type, rather than at compile time.
- **Implementation:**
  - **Virtual Tables (vtables):** When a class is defined with virtual methods, the compiler creates a vtable that maps method calls to the correct implementations.
  - **Late Binding:** This allows for method calls to be resolved at runtime based on the object instance.

Example:

csharp
Copy code
```
public class Shape {
    public virtual void Draw() {
        Console.WriteLine("Drawing a shape");
    }
}
```

```
public class Circle : Shape {
    public override void Draw() {
        Console.WriteLine("Drawing a circle");
    }
}

Shape myShape = new Circle();
myShape.Draw(); // Output: Drawing a circle
```

---

26. What are generics, and how can they be useful in OOP?

- **Definition:** Generics allow you to define classes, interfaces, and methods with a placeholder for the type of data they store or operate on.
- **Benefits:**
    - **Type Safety:** Helps catch type errors at compile time.
    - **Code Reusability:** Allows you to create more flexible and reusable code without sacrificing type safety.
    - **Performance:** Avoids boxing and unboxing when using value types.

Example:

```
csharp
Copy code
public class GenericList<T> {
    private List<T> items = new List<T>();

    public void Add(T item) {
        items.Add(item);
    }

    public T Get(int index) {
        return items[index];
    }
}
```

```
GenericList<int> intList = new GenericList<int>();
intList.Add(1);
intList.Add(2);
Console.WriteLine(intList.Get(1)); // Output: 2
```

---

27. Explain the concept of object composition and its benefits.

- **Definition:** Composition is a design principle where a class is composed of one or more objects from other classes, rather than inheriting from them.
- **Benefits:**
    - **Flexibility:** Allows you to change the behavior of a class by changing its component objects without modifying the class itself.
    - **Reduced Coupling:** Promotes loose coupling between classes, making the system easier to maintain and evolve.
    - **Encapsulation:** Encapsulates the behavior of objects and promotes better organization of code.

Example:

csharp
Copy code
```
public class Engine {
  public void Start() {
    Console.WriteLine("Engine started");
  }
}

public class Car {
  private Engine engine;

  public Car() {
    engine = new Engine();
  }
```

```
    public void Start() {
        engine.Start(); // Delegation
        Console.WriteLine("Car started");
    }
}

Car myCar = new Car();
myCar.Start(); // Output: Engine started
               //         Car started
```

---

28. What is the Liskov Substitution Principle (LSP)? Provide some examples of violation and adherence.

- **Definition:** The Liskov Substitution Principle states that objects of a superclass should be replaceable with objects of a subclass without affecting the correctness of the program.
- **Violation Example:**

csharp
Copy code

```
public class Bird {
    public virtual void Fly() {
        Console.WriteLine("Flying");
    }
}

public class Ostrich : Bird {
    public override void Fly() {
        throw new NotSupportedException("Ostriches cannot fly"); // Violates LSP
    }
}
```

- **Adherence Example:**

csharp

Copy code
```
public class Bird {
  public virtual void MakeSound() {
    Console.WriteLine("Bird sound");
  }
}

public class Sparrow : Bird {
  public override void MakeSound() {
    Console.WriteLine("Chirp");
  }
}

public void SoundTest(Bird bird) {
  bird.MakeSound(); // Works with any Bird subclass
}
```

---

29. What is the dependency inversion principle?

- **Definition:** The Dependency Inversion Principle states that high-level modules should not depend on low-level modules; both should depend on abstractions (interfaces or abstract classes). Additionally, abstractions should not depend on details; details should depend on abstractions.
- **Benefits:** Increases code modularity, reduces coupling, and makes the system easier to test and maintain.

Example:

csharp
Copy code
```
public interface INotificationService {
  void Notify(string message);
}

public class EmailNotification : INotificationService {
  public void Notify(string message) {
      Console.WriteLine($"Email sent: {message}");
  }
}

public class User {
  private INotificationService notificationService;

  public User(INotificationService notificationService) {
      this.notificationService = notificationService;
  }

  public void Register() {
      notificationService.Notify("User registered");
  }
}

// Usage
INotificationService emailService = new EmailNotification();
User user = new User(emailService);
user.Register(); // Output: Email sent: User registered
```

---

30. How can the open/closed principle guide object-oriented design?

- **Definition:** The Open/Closed Principle states that software entities (classes, modules, functions) should be open for extension but closed for modification. This encourages the design of systems that can be easily extended without changing existing code.
- **Benefits:** Facilitates adding new features while minimizing the risk of introducing bugs in existing functionality.

Example:

csharp
Copy code

```
public abstract class Shape {
   public abstract double Area();
}

public class Circle : Shape {
   public double Radius { get; set; }
   public override double Area() {
      return Math.PI * Radius * Radius;
   }
}

public class Rectangle : Shape {
   public double Width { get; set; }
   public double Height { get; set; }
   public override double Area() {
      return Width * Height;
   }
}

// New shape can be added without modifying existing code
public class Triangle : Shape {
   public double Base { get; set; }
   public double Height { get; set; }
   public override double Area() {
      return 0.5 * Base * Height;
   }
}
```

---

31. Describe how the Interface Segregation Principle affects system design.

- **Definition:** The Interface Segregation Principle states that no client should be forced to depend on methods it does not use. Instead of one large interface, multiple smaller interfaces are preferred.

- **Benefits:** Reduces the impact of changes, improves code readability, and increases system modularity.

Example:

csharp
Copy code

```
public interface IAnimal {
   void Eat();
}

public interface IFlyable {
   void Fly();
}

public class Bird : IAnimal, IFlyable {
   public void Eat() {
      Console.WriteLine("Bird is eating");
   }

   public void Fly() {
      Console.WriteLine("Bird is flying");
   }
}

public class Dog : IAnimal {
   public void Eat() {
      Console.WriteLine("Dog is eating");
   }
}

// Dog does not need to implement Fly method
```

---

32. What is a mixin, and how does it differ from traditional inheritance?

- **Definition:** A mixin is a class that provides methods that can be used by other classes but is not considered a base class. Mixins allow behavior to be added to multiple classes without forming a rigid class hierarchy.
- **Difference:** Unlike traditional inheritance, which represents an "is-a" relationship, mixins represent a "has-a" or "can-do" relationship. C# does not natively support mixins, but they can be simulated with interfaces.

Example of mixin-like behavior using interfaces:

csharp
Copy code
```
public interface IFlyable {
    void Fly();
}

public class Bird : IFlyable {
    public void Fly() {
        Console.WriteLine("Bird is flying");
    }
}

public class SuperHero : IFlyable {
    public void Fly() {
        Console.WriteLine("SuperHero is flying");
    }
}
```

---

33. How would you refactor a class that has too many responsibilities?

- **Definition:** Refactoring a class with too many responsibilities involves applying the Single Responsibility Principle (SRP) by breaking the class into smaller, more focused classes.
- **Approach:**
    - Identify the distinct responsibilities.

- Create new classes to handle each responsibility.
- Use composition to delegate tasks among the classes.

Example: Before Refactoring:

csharp

Copy code

```
public class User {
   public void Register() {
      // User registration logic
   }

   public void SendEmailConfirmation() {
      // Email confirmation logic
   }

   public void LogUserAction() {
      // Logging logic
   }
}
```

After Refactoring:

csharp

Copy code

```
public class User {
   public void Register() {
      // User registration logic
   }
}

public class EmailService {
   public void SendEmailConfirmation() {
      // Email confirmation logic
   }
}
```

```
public class Logger {
  public void LogUserAction() {
    // Logging logic
  }
}
```

---

34. Describe a singleton pattern and discuss its pros and cons.

- **Definition:** The Singleton Pattern ensures that a class has only one instance and provides a global point of access to that instance.
- **Pros:**
  - Controlled access to a single instance.
  - Reduces memory usage if instantiation is expensive.
- **Cons:**
  - Can introduce global state, making testing and debugging harder.
  - May lead to tight coupling between classes.

Example:

csharp
Copy code

```
public class Singleton {
  private static Singleton instance;

  private Singleton() { }

  public static Singleton Instance {
    get {
      if (instance == null) {
        instance = new Singleton();
      }
      return instance;
    }
  }
}
```

---

35. What is a factory method, and when should it be used?

- **Definition:** A factory method is a method that returns an instance of a class. It allows subclasses to alter the type of objects that will be created.
- **When to Use:**
  - When the exact type of the object to create isn’ t known until runtime.
  - To encapsulate the instantiation logic of a class.

Example:

csharp
Copy code
```
public abstract class Product {
  public abstract string GetInfo();
}

public class ConcreteProductA : Product {
  public override string GetInfo() {
    return "Product A";
  }
}

public class ConcreteProductB : Product {
  public override string GetInfo() {
    return "Product B";
  }
}

public abstract class Creator {
  public abstract Product FactoryMethod();
}
```

```
public class ConcreteCreatorA : Creator {
   public override Product FactoryMethod() {
      return new ConcreteProductA();
   }
}

public class ConcreteCreatorB : Creator {
   public override Product FactoryMethod() {
      return new ConcreteProductB();
   }
}

// Usage
Creator creator = new ConcreteCreatorA();
Product product = creator.FactoryMethod();
Console.WriteLine(product.GetInfo()); // Output: Product A
```

---

36. Explain the builder pattern and where you might apply it.

- **Definition:** The Builder Pattern is a creational pattern that allows constructing complex objects step by step. It separates the construction of a complex object from its representation.
- **Application:** Useful when an object needs to be created with many optional parameters or when its creation process is complex.

Example:

csharp
Copy code
```
public class Car {
   public string Color { get; set; }
   public string Engine { get; set; }
   public int Wheels { get; set; }
}
```

```
public class CarBuilder {
    private Car car;

    public CarBuilder() {
        car = new Car();
    }

    public CarBuilder SetColor(string color) {
        car.Color = color;
        return this;
    }

    public CarBuilder SetEngine(string engine) {
        car.Engine = engine;
        return this;
    }

    public CarBuilder SetWheels(int wheels) {
        car.Wheels = wheels;
        return this;
    }

    public Car Build() {
        return car;
    }
}

// Usage
Car myCar = new CarBuilder()
    .SetColor("Red")
    .SetEngine("V8")
    .SetWheels(4)
    .Build();
```

---

37. What is the prototype pattern, and how does it relate to OOP?

- **Definition:** The Prototype Pattern is a creational pattern that allows cloning of objects, even complex ones, without needing to know their exact class type.
- **Relation to OOP:** It promotes code reusability and avoids the cost of creating a new instance of an object from scratch.

Example:

```
csharp
Copy code
public abstract class Prototype {
  public abstract Prototype Clone();
}

public class ConcretePrototype : Prototype {
  public string Field { get; set; }

  public override Prototype Clone() {
    return (Prototype)this.MemberwiseClone(); // Shallow copy
  }
}

// Usage
ConcretePrototype prototype = new ConcretePrototype() { Field = "Original" };
ConcretePrototype clone = (ConcretePrototype)prototype.Clone();
clone.Field = "Clone";
Console.WriteLine(prototype.Field); // Output: Original
Console.WriteLine(clone.Field); // Output: Clone
```

---

38. When would you use the Adapter pattern?

- **Definition:** The Adapter Pattern allows incompatible interfaces to work together. It acts as a bridge between two incompatible interfaces.
- **Use Cases:**

- When you want to use an existing class, but its interface does not match the one you need.
- When you want to create a reusable class that cooperates with classes that have different interfaces.

Example:

```
csharp
Copy code
public interface ITarget {
   void Request();
}

public class Adaptee {
   public void SpecificRequest() {
      Console.WriteLine("Specific request");
   }
}

public class Adapter : ITarget {
   private Adaptee adaptee;

   public Adapter(Adaptee adaptee) {
      this.adaptee = adaptee;
   }

   public void Request() {
      adaptee.SpecificRequest();
   }
}

// Usage
ITarget target = new Adapter(new Adaptee());
target.Request(); // Output: Specific request
```

---

39. Can you explain the use of the Decorator pattern?

- **Definition:** The Decorator Pattern allows behavior to be added to individual objects, either statically or dynamically, without affecting the behavior of other objects from the same class.
- **Use Cases:**
  - When you need to add responsibilities to objects without subclassing.
  - When you want to add features to an object at runtime.

Example:

csharp
Copy code
```
public interface ICar {
  string GetDescription();
  double GetCost();
}

public class BasicCar : ICar {
  public string GetDescription() {
    return "Basic Car";
  }

  public double GetCost() {
    return 10000;
  }
}

public abstract class CarDecorator : ICar {
  protected ICar car;

  public CarDecorator(ICar car) {
    this.car = car;
  }

  public abstract string GetDescription();
```

```
    public abstract double GetCost();
}

public class LeatherSeatsDecorator : CarDecorator {
    public LeatherSeatsDecorator(ICar car) : base(car) { }

    public override string GetDescription() {
        return car.GetDescription() + ", Leather Seats";
    }

    public override double GetCost() {
        return car.GetCost() + 2000;
    }
}

// Usage
ICar car = new BasicCar();
car = new LeatherSeatsDecorator(car);
Console.WriteLine(car.GetDescription()); // Output: Basic Car, Leather Seats
Console.WriteLine(car.GetCost()); // Output: 12000
```

---

40. Describe the Observer pattern and a scenario in which you might use it.

- **Definition:** The Observer Pattern defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
- **Use Cases:**
    - When a change in one object requires changing others, and you don’t want tight coupling between them.
    - In event handling systems.

Example:

csharp

Copy code

```
public interface IObserver {
   void Update(string message);
}

public class ConcreteObserver : IObserver {
   private string name;

   public ConcreteObserver(string name) {
      this.name = name;
   }

   public void Update(string message) {
      Console.WriteLine($"{name} received: {message}");
   }
}

public class Subject {
   private List<IObserver> observers = new List<IObserver>();

   public void Attach(IObserver observer) {
      observers.Add(observer);
   }

   public void Notify(string message) {
      foreach (var observer in observers) {
         observer.Update(message);
      }
   }
}

// Usage
Subject subject = new Subject();
IObserver observer1 = new ConcreteObserver("Observer 1");
IObserver observer2 = new ConcreteObserver("Observer 2");
```

```
subject.Attach(observer1);
subject.Attach(observer2);
subject.Notify("Event has occurred!");
// Output:
// Observer 1 received: Event has occurred!
// Observer 2 received: Event has occurred!
```

41. What are the advantages of using the Command pattern?

- **Definition:** The Command Pattern encapsulates a request as an object, thereby allowing for parameterization of clients with queues, requests, and operations.
- **Advantages:**
  - **Decoupling:** Separates the sender of a request from the object that handles it.
  - **Undo/Redo functionality:** Commands can be stored and executed, allowing easy undo and redo operations.
  - **Logging:** Commands can be logged for auditing or debugging purposes.
  - **Batch operations:** Commands can be grouped together for batch processing.
  - **Flexible operations:** New commands can be added without changing existing code.

Example:

csharp
Copy code

```
public interface ICommand {
   void Execute();
}

public class Light {
   public void TurnOn() {
       Console.WriteLine("Light is ON");
   }

   public void TurnOff() {
       Console.WriteLine("Light is OFF");
   }
}

public class LightOnCommand : ICommand {
   private Light light;

   public LightOnCommand(Light light) {
       this.light = light;
   }

   public void Execute() {
       light.TurnOn();
   }
}

public class LightOffCommand : ICommand {
   private Light light;

   public LightOffCommand(Light light) {
       this.light = light;
   }

   public void Execute() {
       light.TurnOff();
   }
}

// Usage
Light light = new Light();
ICommand lightOn = new LightOnCommand(light);
ICommand lightOff = new LightOffCommand(light);
```

```
lightOn.Execute(); // Output: Light is ON
lightOff.Execute(); // Output: Light is OFF
```

---

42. How does the Strategy pattern provide flexibility in objects?

- **Definition:** The Strategy Pattern defines a family of algorithms, encapsulates each one, and makes them interchangeable. It lets the algorithm vary independently from clients that use it.
- **Flexibility:** Clients can choose which algorithm to use at runtime, allowing dynamic behavior changes without altering the context.

Example:

csharp
Copy code
```
public interface IStrategy {
   int Execute(int a, int b);
}

public class AdditionStrategy : IStrategy {
   public int Execute(int a, int b) {
      return a + b;
   }
}

public class SubtractionStrategy : IStrategy {
   public int Execute(int a, int b) {
      return a - b;
   }
}

public class Context {
   private IStrategy strategy;
```

```
    public void SetStrategy(IStrategy strategy) {
        this.strategy = strategy;
    }

    public int ExecuteStrategy(int a, int b) {
        return strategy.Execute(a, b);
    }
}

// Usage
Context context = new Context();
context.SetStrategy(new AdditionStrategy());
Console.WriteLine(context.ExecuteStrategy(5, 3)); // Output: 8

context.SetStrategy(new SubtractionStrategy());
Console.WriteLine(context.ExecuteStrategy(5, 3)); // Output: 2
```

---

43. What are some common OOP design anti-patterns?

- **Definition:** Anti-patterns are ineffective solutions that are counterproductive to the goals of design.
- **Common Anti-Patterns:**
  - **God Object:** A class that knows too much or does too much, violating SRP (Single Responsibility Principle).
  - **Spaghetti Code:** Unstructured code that is difficult to maintain and follow due to lack of organization.
  - **Copy-Paste Programming:** Duplicating code across the codebase, making maintenance difficult.
  - **Feature Creep:** Adding unnecessary features to the system that complicate it without adding real value.
  - **Golden Hammer:** Relying too heavily on a single tool or method for all problems, regardless of suitability.

44. How do you ensure that your objects are properly encapsulated?

- **Definition:** Encapsulation is the bundling of data and methods that operate on that data within one unit, restricting access to some of the object's components.
- **Techniques:**
  - Use **private** and **protected** access modifiers to hide data.
  - Provide **public** methods (getters/setters) to access or modify private data.
  - Keep the internal implementation details hidden and expose only what is necessary.

Example:

csharp

Copy code

```
public class BankAccount {
  private double balance;

  public BankAccount(double initialBalance) {
    balance = initialBalance;
  }

  public double GetBalance() {
    return balance;
  }

  public void Deposit(double amount) {
    if (amount > 0) {
      balance += amount;
    }
  }
```

```
    public void Withdraw(double amount) {
        if (amount > 0 && amount <= balance) {
            balance -= amount;
        }
    }
}

// Usage
BankAccount account = new BankAccount(100);
account.Deposit(50);
Console.WriteLine(account.GetBalance()); // Output: 150
```

---

45. Name some techniques for reducing coupling between classes.

- **Definition:** Coupling refers to the degree of direct knowledge that one class has of another. Lower coupling improves maintainability and flexibility.
- **Techniques:**
  - **Interfaces:** Use interfaces to define contracts without dictating implementation.
  - **Dependency Injection:** Provide dependencies externally rather than hardcoding them within the class.
  - **Events and Delegates:** Use events to notify other classes without them needing to know each other directly.
  - **Service Locator Pattern:** Use a service locator to provide dependencies at runtime without direct coupling.
  - **Abstract Classes:** Use abstract classes to provide a common base without forcing implementation details.

---

46. How does immutability help in object-oriented design, and how can it be implemented?

- **Definition:** Immutability refers to objects whose state cannot be modified after they are created. This can simplify design and make objects easier to manage.
- **Benefits:**
  - **Thread Safety:** Immutable objects are inherently thread-safe.
  - **Easier reasoning:** They simplify the understanding of object state throughout the application.
  - **Functional programming:** Aligns with functional programming principles.

Implementation Example:

```
csharp
Copy code
public class ImmutablePoint {
  public int X { get; }
  public int Y { get; }

  public ImmutablePoint(int x, int y) {
    X = x;
    Y = y;
  }

  public ImmutablePoint WithX(int newX) {
    return new ImmutablePoint(newX, Y);
  }

  public ImmutablePoint WithY(int newY) {
    return new ImmutablePoint(X, newY);
  }
}

// Usage
var point = new ImmutablePoint(1, 2);
var newPoint = point.WithX(3);
```

```
Console.WriteLine(point.X); // Output: 1
Console.WriteLine(newPoint.X); // Output: 3
```

---

47. What tools or techniques would you use to document an object-oriented design?

- **Documentation Tools:**
  - **UML Diagrams:** Use UML diagrams (class diagrams, sequence diagrams) to visualize relationships and interactions.
  - **API Documentation Tools:** Use tools like Swagger, DocFX, or Javadoc to generate documentation from comments.
  - **Code Comments:** Write meaningful comments within the code to describe purpose and usage.
  - **Design Patterns Documentation:** Use design pattern catalogs or wikis to explain patterns used in the design.

---

48. How do you address circular dependencies in an OOP system?

- **Definition:** Circular dependencies occur when two or more classes depend on each other, creating a loop that can lead to issues in the code.
- **Strategies:**
  - **Refactor:** Break the dependency by introducing an interface or a third class that handles the dependencies.
  - **Dependency Injection:** Use dependency injection frameworks to manage dependencies and break circular references.
  - **Event-Driven Approach:** Use events and callbacks to allow classes to communicate without directly depending on one another.

---

49. Explain how to apply unit testing to object-oriented code.

- **Definition:** Unit testing involves testing individual components (units) of the code in isolation to ensure they work as intended.
- **Approach:**
    - Use testing frameworks like MSTest, NUnit, or xUnit.
    - Mock dependencies to isolate the unit being tested using mocking frameworks like Moq.
    - Write tests that cover different scenarios, including edge cases.
    - Use assertions to verify expected outcomes.

Example:

```
csharp
Copy code
[TestClass]
public class BankAccountTests {
  [TestMethod]
  public void Deposit_ShouldIncreaseBalance() {
    // Arrange
    var account = new BankAccount(100);

    // Act
    account.Deposit(50);

    // Assert
    Assert.AreEqual(150, account.GetBalance());
  }
}
```

---

50. What strategies can be used to safely refactor legacy object-oriented code?

- **Strategies:**

- **Write Unit Tests:** Before refactoring, write unit tests to ensure existing functionality is preserved.
- **Incremental Refactoring:** Make small, incremental changes rather than large-scale refactorings to reduce risk.
- **Use Version Control:** Use version control systems to keep track of changes and allow easy rollbacks.
- **Continuous Integration:** Integrate changes frequently and run tests automatically to catch issues early.
- **Code Review:** Use peer reviews to catch potential issues in the refactoring process.

---

51. How can the principles of OOP help in achieving a modular and maintainable codebase?

- **Principles:**
  - **Encapsulation:** Limits exposure of internal states, making changes easier without affecting other parts.
  - **Inheritance:** Promotes code reuse and allows for extensions without modification.
  - **Polymorphism:** Provides flexibility in using objects, allowing different implementations to be treated uniformly.
  - **Abstraction:** Simplifies complex systems by hiding unnecessary details, focusing on essential features.

---

52. How do you balance the use of OOP principles with performance considerations in a system design?

- **Balance Strategies:**

    - **Profile Performance:** Use profiling tools to identify bottlenecks caused by OOP abstractions.
    - **Selective Abstraction:** Apply OOP principles where they provide clear benefits; avoid unnecessary abstraction.
    - **Optimize Critical Paths:** Optimize performance-critical sections of code without sacrificing clarity in less critical areas.
    - **Measure Trade-offs:** Always measure the trade-offs between maintainability, readability, and performance, making informed decisions.

53. What is the difference between an abstract class and an interface in C#?

- **Abstract Class:**
    - Can have both abstract methods (without implementation) and concrete methods (with implementation).
    - Can have fields, constructors, and access modifiers.
    - Supports code reuse through shared functionality.
- **Interface:**
    - Can only declare methods, properties, events, and indexers.
    - Cannot contain any implementation (C# 8.0 introduced default interface methods, but it's limited).
    - A class can implement multiple interfaces, supporting multiple inheritance.

Example:

csharp
Copy code
```
public abstract class Animal {
  public abstract void Speak();
  public void Sleep() {
    Console.WriteLine("Sleeping...");
  }
}

public interface IMammal {
  void FeedYoung();
}

public class Dog : Animal, IMammal {
  public override void Speak() {
    Console.WriteLine("Bark!");
  }
  public void FeedYoung() {
    Console.WriteLine("Feeding puppies.");
  }
}
```

---

54. How do you implement the Singleton pattern in C#?

- **Definition:** The Singleton Pattern ensures a class has only one instance and provides a global access point to that instance.
- **Implementation:**
  - Use a private static variable to hold the single instance.
  - Provide a public static method to return the instance, creating it if it doesn't exist.

Example:

csharp

Copy code

```
public class Singleton {
  private static Singleton instance;

  private Singleton() { }

  public static Singleton Instance {
        get {
            if (instance == null) {
                instance = new Singleton();
            }
            return instance;
        }
    }

    public void DoSomething() {
        Console.WriteLine("Doing something...");
    }
}

// Usage
Singleton.Instance.DoSomething();
```

---

55. What is the purpose of the Adapter pattern, and how is it implemented?

- **Definition:** The Adapter Pattern allows incompatible interfaces to work together by wrapping an existing class with a new interface.
- **Purpose:** It enables classes to work together that couldn’ t otherwise due to incompatible interfaces.

Example:

csharp
Copy code
```
public interface ITarget {
    void Request();
}

public class Adaptee {
    public void SpecificRequest() {
        Console.WriteLine("Called SpecificRequest.");
    }
}

public class Adapter : ITarget {
    private Adaptee adaptee;

    public Adapter(Adaptee adaptee) {
        this.adaptee = adaptee;
    }

    public void Request() {
        adaptee.SpecificRequest();
    }
}

// Usage
ITarget target = new Adapter(new Adaptee());
target.Request(); // Output: Called SpecificRequest.
```

---

56. What is the purpose of the Builder pattern, and when would you use it?

- **Definition:** The Builder Pattern separates the construction of a complex object from its representation, allowing the same construction process to create different representations.
- **Purpose:** It' s useful when an object needs to be created with many optional parameters or complex construction logic.

Example:

csharp
Copy code
```
public class Car {
    public string Model { get; private set; }
    public string Color { get; private set; }
    public int Year { get; private set; }

    public class Builder {
        private string model;
        private string color;
        private int year;

        public Builder SetModel(string model) {
            this.model = model;
            return this;
        }

        public Builder SetColor(string color) {
            this.color = color;
            return this;
        }

        public Builder SetYear(int year) {
            this.year = year;
            return this;
        }

        public Car Build() {
            return new Car { Model = model, Color = color, Year = year };
        }
    }
}

// Usage
Car car = new Car.Builder()
    .SetModel("Tesla")
    .SetColor("Red")
    .SetYear(2024)
    .Build();
```

---

57. What is the Repository pattern and how does it promote separation of concerns?

- **Definition:** The Repository Pattern is a design pattern that mediates data from and to the domain and data mapping layers, allowing for a separation of concerns.
- **Benefits:**
    - It abstracts data access logic and provides a more object-oriented view of the data.
    - It promotes cleaner code by separating the business logic from data access logic.

Example:

csharp
Copy code

```
public interface IRepository<T> {
   void Add(T entity);
   T Get(int id);
   IEnumerable<T> GetAll();
}

public class User {
   public int Id { get; set; }
   public string Name { get; set; }
}

public class UserRepository : IRepository<User> {
   private List<User> users = new List<User>();

   public void Add(User user) {
      users.Add(user);
   }

   public User Get(int id) {
        return users.FirstOrDefault(u => u.Id == id);
    }

    public IEnumerable<User> GetAll() {
        return users;
    }
}

// Usage
UserRepository userRepository = new UserRepository();
userRepository.Add(new User { Id = 1, Name = "John" });
User user = userRepository.Get(1);
```

---

58. What is the role of the Factory Method pattern?

- **Definition:** The Factory Method Pattern defines an interface for creating an object but allows subclasses to alter the type of objects that will be created.
- **Purpose:** It promotes loose coupling by eliminating the need to specify the exact class of the object that will be created.

Example:

csharp
Copy code
```
public abstract class Product {
    public abstract string GetName();
}

public class ConcreteProductA : Product {
    public override string GetName() {
        return "Product A";
    }
}

public class ConcreteProductB : Product {
    public override string GetName() {
        return "Product B";
    }
}

public abstract class Creator {
    public abstract Product FactoryMethod();
}

public class ConcreteCreatorA : Creator {
    public override Product FactoryMethod() {
        return new ConcreteProductA();
    }
}

public class ConcreteCreatorB : Creator {
    public override Product FactoryMethod() {
        return new ConcreteProductB();
    }
}

// Usage
Creator creator = new ConcreteCreatorA();
Product product = creator.FactoryMethod();
Console.WriteLine(product.GetName()); // Output: Product A
```

---

59. How do you implement the Observer pattern in C#?

- **Definition:** The Observer Pattern defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
- **Use Case:** It's commonly used in event handling systems.

Example:

csharp

Copy code

```
public interface IObserver {
    void Update(string message);
}

public interface ISubject {
    void Attach(IObserver observer);
    void Detach(IObserver observer);
    void Notify();
}

public class ConcreteSubject : ISubject {
    private List<IObserver> observers = new List<IObserver>();
    private string state;

    public string State {
        get { return state; }
        set {
            state = value;
            Notify();
        }
    }

    public void Attach(IObserver observer) {
        observers.Add(observer);
    }

    public void Detach(IObserver observer) {
        observers.Remove(observer);
    }

    public void Notify() {
        foreach (var observer in observers) {
            observer.Update(state);
        }
    }
}

public class ConcreteObserver : IObserver {
    private string name;

    public ConcreteObserver(string name) {
        this.name = name;
    }

    public void Update(string message) {
        Console.WriteLine($"{name} received: {message}");
    }
}

// Usage
ConcreteSubject subject = new ConcreteSubject();
ConcreteObserver observer1 = new ConcreteObserver("Observer 1");
ConcreteObserver observer2 = new ConcreteObserver("Observer 2");

subject.Attach(observer1);
subject.Attach(observer2);

subject.State = "New State!";
// Output:
// Observer 1 received: New State!
// Observer 2 received: New State!
```

---

60. What is the role of the Mediator pattern in OOP?

- **Definition:** The Mediator Pattern defines an object that encapsulates how a set of objects interact, promoting loose coupling by keeping objects from referring to each other explicitly.
- **Purpose:** It centralizes complex communications and control between related objects.

Example:

csharp
Copy code
```
public interface IMediator {
    void Notify(object sender, string ev);
}

public class ConcreteMediator : IMediator {
    private ComponentA componentA;
    private ComponentB componentB;

    public ConcreteMediator(ComponentA a, ComponentB b) {
        componentA = a;
        componentB = b;

        componentA.SetMediator(this);
        componentB.SetMediator(this);
    }

    public void Notify(object sender, string ev) {
        if (sender == componentA && ev == "A") {
            Console.WriteLine("Mediator reacts on A's event.");
            componentB.DoSomething();
        }
        if (sender == componentB && ev == "B") {
            Console.WriteLine("Mediator reacts on B's event.");
            componentA.DoSomething();
        }
    }
}

public class ComponentA {
    private IMediator mediator;

    public void SetMediator(IMediator mediator) {
        this.mediator = mediator;
    }

    public void DoSomething() {
        Console.WriteLine("Component A does something.");
        mediator.Notify(this, "A");
    }
}

public class ComponentB {
    private IMediator mediator;

    public void SetMediator(IMediator mediator) {
        this.mediator = mediator;
    }

    public void DoSomething() {
        Console.WriteLine("Component B does something.");
        mediator.Notify(this, "B");
    }
}

// Usage
ComponentA a = new ComponentA();
ComponentB b = new ComponentB();
ConcreteMediator mediator = new ConcreteMediator(a, b);

a.DoSomething(); // Output: Component A does something. Mediator reacts on A's event.
Component B does something.
```