Object-Oriented Programming

PRG 281

Daniel Mattheus Johannes Adendorff 577624

Contents

[Question 1 2](#_Toc143176587)

[a) Inheritance: 2](#_Toc143176588)

[Code for inheritance 3](#_Toc143176589)

[b) Abstraction: 4](#_Toc143176590)

[Abstraction code example 5](#_Toc143176591)

[c) Encapsulation: 6](#_Toc143176592)

[Encapsulation code example 7](#_Toc143176593)

[d) Polymorphis: 8](#_Toc143176594)

[Polymorphis code example 9](#_Toc143176595)

[Question 2 10](#_Toc143176596)

[1. Method Overriding: 10](#_Toc143176597)

[Method Overriding code Example 11](#_Toc143176598)

[2. Interfaces: 12](#_Toc143176599)

[Interfaces code Example 13](#_Toc143176600)

# Question 1

Explain the following Object-Oriented Programming (OOP) principles with examples.   
a) Inheritance   
b) Abstraction   
c) Encapsulation   
d) Polymorphis

## a) Inheritance:

Inheritance is used to create a new class that is based on an existing class. The new class inherits properties and methods from the existing class, promoting code reusability.

Inheritance is a mechanism in object-oriented programming where a new class can inherit properties and behaviours from an existing class Inheritance promotes code reusability and establishes a hierarchical relationship among classes. The derived class can enhance or modify the inherited properties and methods and may also introduce new ones

Example Explanation: Inheritance allows you to model relationships between real-world entities. In the given example of vehicles, the Vehicle class represents a common set of attributes and behaviours that all vehicles share, such as a brand. The Car and Bicycle classes then inherit these common attributes and behaviours, and they can add their own specific properties while still having access to the start engine method from the base class.

### Code for inheritance

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace inheritance

{

class Vehicle

{

public string Brand {get; set;}

public void StartEngine()

{

Console.WriteLine($"The {Brand} vehicle's engine is starting.");

}

}

class Car : Vehicle

{

public string Model {get; set;}

}

class Bicycle : Vehicle

{

public string Type {get; set;}

}

class Program

{

static void Main()

{

Car car=new Car {Brand="Toyota", Model="Camry"};

Bicycle bicycle=new Bicycle {Brand="Trek", Type="Mountain"};

car.StartEngine();

bicycle.StartEngine();

}

}

}

## b) Abstraction:

Abstraction allows you to define a common interface while hiding the underlying implementation details in derived classes.

Abstraction is the process of simplifying complex reality by modelling classes based on essential characteristics and ignoring unnecessary details. It involves defining abstract classes or interfaces that provide a blueprint for other classes to follow. Abstraction allows you to create a higher-level view of an object's properties and behaviours, hiding the implementation details.

Example Explanation: Abstraction is particularly useful for designing systems where you want to define a common interface for related classes. In the geometric shape example, the Shape class is an abstraction that defines the contract for calculating the area of various shapes. The Rectangle and Circle classes implement this contract by providing specific implementations for the Calculate Area method. This allows you to work with shapes generically without concerning yourself with the specific calculations each shape performs

### Abstraction code example

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace Abstraction

{

abstract class Shape

{

public abstract double CalculateArea();

}

class Rectangle : Shape

{

public double Width {get; set;}

public double Height {get; set;}

public override double CalculateArea()

{

return Width \* Height;

}

}

class Circle : Shape

{

public double Radius { get; set; }

public override double CalculateArea()

{

return Math.PI \* Radius \* Radius;

}

}

class Program

{

static void Main()

{

Rectangle rectangle = new Rectangle {Width=5, Height=3};

Circle circle = new Circle {Radius=2};

Console.WriteLine($"Rectangle Area: {rectangle.CalculateArea()}");

Console.WriteLine($"Circle Area: {circle.CalculateArea()}");

}

}

}

## c) Encapsulation:

Encapsulation restricts the access to an object's data and methods, allowing controlled interaction with the object.

Encapsulation is the concept of bundling data and methods that operate on the data into a single unit, called a class. It helps in controlling access to the internal state of an object, promoting data integrity and security. Encapsulation hides the implementation details of an object and exposes only the necessary interface to interact with it.

Example Explanation: Encapsulation is essential for creating well-structured and maintainable code. In the bank account example, the Bank Account class encapsulates the account number and balance attributes. The acess to these attributes is controlled through methods like Deposit, Withdraw, and Get Balance. This ensures that the account's balance can only be modified using valid operations and prevents direct manipulation of the attributes from outside the class.

### Encapsulation code example

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace Encapsulation

{

class BankAccount

{

private int AccountNumber {get; set;}

private decimal Balance {get; set;}

public BankAccount(int accountNumber, decimal balance)

{

AccountNumber=accountNumber;

Balance=balance;

}

public void Deposit(decimal amount)

{

if (amount > 0)

{

Balance+=amount;

}

}

public void Withdraw(decimal amount)

{

if (amount > 0 && amount <= Balance)

{

Balance-=amount;

}

}

public decimal GetBalance()

{

return Balance;

}

}

class Program

{

static void Main()

{

BankAccount account=new BankAccount(accountNumber: 12345, balance: 339375634728);

account.Deposit(500000);

account.Withdraw(2000000);

Console.WriteLine($"Account Balance: {account.GetBalance()}");

}

}

}

## d) Polymorphis:

Polymorphis allows objects of different classes to be treated as objects of a common base class, enabling flexibility in method invocation.

Polymorphis allows objects of different classes to be treated as objects of a common base class. It enables the flexibility to use a single interface to represent multiple types of objects. Polymorphism is often achieved through method overriding and interfaces.

Polymorphis enhances code flexibility and extensibility. In the animal sounds example, the Animal class defines a virtual Make Sound method. The derived classes Dog, Cat, and Duck override this method to provide their own unique implementations. The Animal Sounds function demonstrates polymorphism by accepting objects of diferent derived classes and invoking the Make Sound method. This enables you to write code that can work with various types of animals without knowing their specific implementations.

These principles collectively form the foundation of object-oriented programming, enabling developers to create modular, maintainable, and extensible software systems. Each principle contributes to different aspects of software design and development, and understanding how they work together is crucial for writing effective and elegant object-oriented code.

### Polymorphis code example

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace Polymorphis

{

using System;

class Animal

{

public virtual string MakeSound()

{

return "Some sound";

}

}

class Dog : Animal

{

public override string MakeSound()

{

return "Woof!";

}

}

class Cat : Animal

{

public override string MakeSound()

{

return "Meow!";

}

}

class Duck : Animal

{

public override string MakeSound()

{

return "Quack!";

}

}

class Program

{

static void AnimalSounds(Animal animal)

{

Console.WriteLine(animal.MakeSound());

}

static void Main()

{

Animal dog=new Dog();

Animal cat=new Cat();

Animal duck=new Duck();

AnimalSounds(dog);

AnimalSounds(cat);

AnimalSounds(duck);

}

}

}

# Question 2

How do you implement polymorphism in Object Oriented Programming? Use examples to support your arguments.

## 1. Method Overriding:

Method overriding is a key feature of polymorphism that allows a subclass to provide a specific implementation for a method that is already defined in its superclass. The overridden method in the subclass has the same name, return type, and parameters as the method in the superclass. In this example we consider a scenario where different animals make different sounds. We can use method overriding to achieve polymorphism:

In the example, the MakeSound method is overriddenin the Dog and Cat classes. When the method is called on instances of these classes, the appropriate overridden implementation is executed based on the actual object type.

### Method Overriding code Example

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace Method Overriding

{

using System;

class Animal

{

public virtual void MakeSound()

{

Console.WriteLine("Some sound");

}

}

class Dog : Animal

{

public override void MakeSound()

{

Console.WriteLine("Woof!");

}

}

class Cat : Animal

{

public override void MakeSound()

{

Console.WriteLine("Meow!");

}

}

class Program

{

static void Main()

{

Animal dog=new Dog();

Animal cat=new Cat();

dog.MakeSound();

cat.MakeSound();

}

}

}

## 2. Interfaces:

An interface defines a contract that a class must adhere to. It specifies a set of method signatures that a class implementing the interface must provide. Interfaces allow different classes to share a common set of behaviours, promoting polymorphism. In the example I consider a scenario where different shapes need to calculate their areas. We can use interfaces to achieve polymorphism:

In this example, the IShape interface defines the CalculateArea method. Both the Rectangle and Circle classes implement this interface and provide their own implementations for calculating the area. The Main method can treat instances of both classes as IShapeobjects and call the CalculateArea method, demonstrating polymorphism.

### Interfaces code Example

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace Interfaces

{

using System;

interface IShape

{

double CalculateArea();

}

class Rectangle : IShape

{

public double Width { get; set; }

public double Height { get; set; }

public double CalculateArea()

{

return Width \* Height;

}

}

class Circle : IShape

{

public double Radius { get; set; }

public double CalculateArea()

{

return Math.PI \* Radius \* Radius;

}

}

class Program

{

static void Main()

{

IShape rectangle = new Rectangle { Width=5, Height=3 };

IShape circle = new Circle { Radius=2 };

Console.WriteLine($"Rectangle Area:{rectangle.CalculateArea()}");

Console.WriteLine($"Circle Area:{circle.CalculateArea()}");

}

}

}

In both examples, polymorphism allows us to write code that can work with different types of objects in a unified manner, enhancing code flexibility and maintainability.