Exploring abstract, virtual, override, and sealed to Implement Polymorphism in Csharp

Nagib Sabbag Filho

Leaders.Tec.Br, 1(18), ISSN: 2966-263X, 2024.

e-mail: profnagib.filho@fiap.com.br

DOI: https://doi.org/10.5281/zenodo.14030916

PermaLink: https://leaders.tec.br/article/d3edc4

Nov 04 2024

Abstract:

The article discusses the concept of polymorphism in C#, a pillar of object-oriented programming, allowing methods with the same name to behave differently in different classes. It explores the use of the keywords `abstract`, `virtual`, `override`, and `sealed` to implement polymorphism, illustrating with practical examples.

Key words:

polymorphism, C#, abstract, virtual, override, sealed, classes, inheritance, methods, interfaces, encapsulation, object-oriented programming, software design, extensibility, restriction, implementation, abstract methods, virtual methods, sealed classes, flexibility, reusable code

Introduction to Polymorphism in C#

Polymorphism is one of the central concepts of object-oriented programming, allowing methods with the same name to behave differently in different classes. In C#, polymorphism can be implemented through inheritance and interfaces, using keywords such as abstract, virtual, override, and sealed. In this article, we will explore how these keywords are used to effectively implement polymorphism in C#, as well as discuss the importance of these concepts in building robust and scalable systems.

Abstract Keyword

The abstract keyword is used to declare classes and methods that are incomplete and must be implemented in derived classes. A class marked as abstract cannot be instantiated directly, and its primary purpose is to serve as a base for other classes. This is essential when we want to ensure that certain functionalities are implemented in subclasses. Let’s see a practical example:

```
abstract class Animal
{
    public abstract void MakeSound();
}

class Dog : Animal
{
    public override void MakeSound()
    {
        Console.WriteLine("Woof Woof");
    }
}
```

```
class Cat : Animal
{
    public override void MakeSound()
    {
        Console.WriteLine("Meow");
    }
}
```

In this example, the Animal class is declared as abstract and has a MakeSound method that is also abstract. The Dog and Cat classes inherit from Animal and implement the MakeSound method in different ways, demonstrating the concept of polymorphism. This means that when calling MakeSound on an object of type Animal, the sound corresponding to the specific type of animal will be displayed, showing how polymorphism allows the same method to behave differently.

Using the Virtual Keyword

The virtual keyword allows a method in a base class to have a default implementation but can be overridden in a derived class. This is useful when you want to provide a default behavior but still allow subclasses to modify that behavior as needed. For example:

```
class Vehicle
{
    public virtual void ShowInfo()
    {
        Console.WriteLine("This is a vehicle.");
    }
}

class Car : Vehicle
{
    public override void ShowInfo()
    {
        Console.WriteLine("This is a car.");
    }
}

class Motorcycle : Vehicle
{
    public override void ShowInfo()
    {
        Console.WriteLine("This is a motorcycle.");
    }
}
```

In this example, the Vehicle class has a ShowInfo method that is declared as virtual. The Car and Motorcycle classes override this method, providing specific implementations. This allows code that uses the Vehicle class to call the method and obtain specific information about the type of vehicle. The flexibility of polymorphism is evidenced here, as an object of type Vehicle can be treated as a Car or Motorcycle, and the displayed information will reflect the actual type of the object.

Implementing Override to Redefine Behavior

The override keyword is used in a derived class to implement a method that was declared as virtual or abstract in the

base class. This allows you to redefine the behavior of a method, adapting it to the specific needs of the derived class. Let’s see an example:

```
class Employee
{
    public virtual void CalculateSalary()
    {
        Console.WriteLine("Base salary.");
    }
}

class Manager : Employee
{
    public override void CalculateSalary()
    {
        Console.WriteLine("Manager salary with bonus.");
    }
}

class Intern : Employee
{
    public override void CalculateSalary()
    {
        Console.WriteLine("Intern salary.");
    }
}
```

In the example above, the Employee class has a CalculateSalary method that is virtual. The Manager and Intern classes override this method, implementing their own salary calculation logic. Thus, when an object of type Employee is treated as an Employee, the appropriate method is called according to the actual type of the object. This is a clear example of how polymorphism allows different subclasses to have distinct behaviors, even if they share the same method signature in the base class.

Using Sealed to Restrict Inheritance

The sealed keyword is used to prevent a class from being inherited. This is useful when you want to ensure that the behavior of a class cannot be altered by subclasses. Let’s see an example:

```
class BankAccount
{
    public virtual void Withdraw(decimal amount)
    {
        Console.WriteLine($"Withdrawal of {amount} completed.");
    }
}

sealed class SavingsAccount : BankAccount
{
    public override void Withdraw(decimal amount)
    {
        Console.WriteLine($"Withdrawal of {amount} completed from the savings account."
);
    }
}
```

```
// The following class cannot inherit from SavingsAccount
// class SpecialAccount : SavingsAccount { }
```

In the example above, the SavingsAccount class is marked as sealed, which means it cannot be inherited. This ensures that the implementation of the Withdraw method in the SavingsAccount class will not be altered by subclasses, maintaining the integrity of the class's behavior. Using sealed is a best practice when you want to protect critical logic within a class and avoid unwanted modifications that could arise through inheritance.

Complete Example of Polymorphism in Action

Now let’s put all of this together in a more complex example that uses the keywords abstract, virtual, override, and sealed in a practical scenario. In this example, we will create a structure to calculate the areas of different geometric shapes, demonstrating the effectiveness of polymorphism in C#.

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

class Rectangle : Shape
{
    private double width;
    private double height;

    public Rectangle(double width, double height)
    {
        this.width = width;
        this.height = height;
    }

    public override double CalculateArea()
    {
        return width * height;
    }
}

class Circle : Shape
{
    private double radius;

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

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

class AreaCalculator
{
    public double CalculateTotalArea(Shape[] shapes)
    {
        double totalArea = 0;
```

```
        foreach (var shape in shapes)
        {
            totalArea += shape.CalculateArea();
        }
        return totalArea;
    }
}

class Program
{
    static void Main(string[] args)
    {
        Shape[] shapes = new Shape[]
        {
            new Rectangle(5, 10),
            new Circle(7)
        };

        AreaCalculator calculator = new AreaCalculator();
        double totalArea = calculator.CalculateTotalArea(shapes);
        Console.WriteLine($"Total area: {totalArea}");
    }
}
```

In this example, we have a Shape class that is abstract and defines a CalculateArea method. The Rectangle and Circle classes inherit from Shape and implement the CalculateArea method in different ways. The AreaCalculator class can calculate the total area of different shapes, demonstrating the true power of polymorphism. The CalculateTotalArea method accepts an array of shapes, and regardless of the type of shape being processed, the appropriate method to calculate the area is called, highlighting the flexibility of polymorphism.

Benefits of Polymorphism in C#

Polymorphism in C# offers a range of benefits that are crucial for modern software development:

• Flexibility: Allows the same code to operate on different types of objects, making it easier to extend and maintain software.

• Code Reusability: Reduces code duplication, as common behavior can be defined in a base class and reused in derived classes.

• Ease of Maintenance: Changes in the logic of a base class automatically propagate to derived classes, reducing the risk of errors.

• Decoupling: Promotes a cleaner, decoupled design where parts of the code depend on abstractions rather than concrete implementations.

Final Considerations

The use of the abstract, virtual, override, and sealed keywords in C# provides a powerful set of tools for implementing polymorphism. These tools enable developers to create flexible and extensible systems where behavior can be modified without affecting existing code. Understanding and efficiently using these keywords is essential for any C# programmer who wishes to master object-oriented programming. By applying these concepts, you will be able to create more robust applications that are prepared for future evolution.

References

• MICROSOFT. C# Programming Guide. Available at: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/inside-a-program/programming-guide-using-abstract-classes. Accessed on: October 20, 2023.

• MICROSOFT. Polymorphism in C#. Available at: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/interfaces/polymorphism. Accessed on: October 20, 2023.

• MICROSOFT. Sealed Classes and Methods in C#. Available at: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/sealed-classes-and-sealed-methods. Accessed on: October 20, 2023.

Nagib is a University Professor and Tech Manager. He has a track record of achievements in technical and agile certifications, including MCSD, MCSA, and PSM1. He holds a postgraduate degree in IT Management from SENAC and an MBA in Software Technology from USP. Nagib has completed extension programs at MIT and the University of Chicago. Other achievements include authoring a peer-reviewed article on chatbots, presented at the University of Barcelona.