20CYS312 - Principles of Programming Languages
Exploring Programming Paradigms

Assignment-01

Presented by Arjun C Santhosh
CB.EN.U4CYS21010
TIFAC-CORE in Cyber Security
Amrita Vishwa Vidyapeetham, Coimbatore Campus

Feb 2024

Outline

Programming Paradigms

Figure: C# (Object Oriented Programming)

Figure: Spring Framework (Aspect-oriented programming)

Object-oriented programming

- what is Object-oriented programming
- Object-Oriented Programming (OOP) is a programming paradigm based on the concept of "objects," which can encapsulate data in the form of fields (attributes or properties) and code in the form of procedures (methods or functions). The primary goals of Object- Oriented Programming are to organize and structure code in a way that promotes modularity, reusability, and extensibility
- Languages that support Object-oriented programming are C++ , C# , Java , Python ,JavaScript etc

Core Concepts of Object-oriented programming

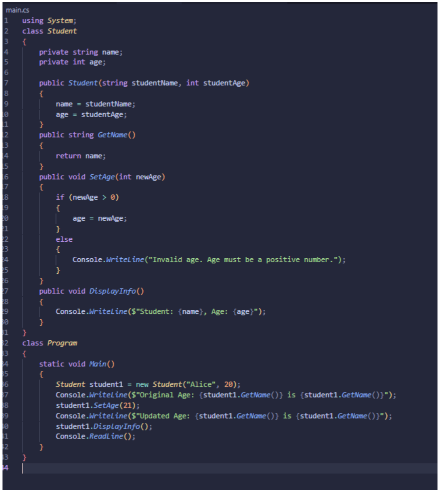
- Objects
  - Objects are instance of classes
- Classes
  - Classes are blueprints or templates for creating objects. They define the structure and behavior that objects instantiated from them will have
- Encapsulation
  - Encapsulation is bundling the attributes and functions that operate on the data into a single unit or class, restricting direct access to it
- Inheritance
  - Inheritance allows a class to inherit properties and methods from a parent class . This allows code reuse
- Abstraction
  - simplifying complex systems by modeling classes based on essential features, hiding unnecessary details, and providing a high-level, generalized interface for interacting with objects
- Polymorphism
  - polymorphism describes the concept that you can access objects of different types through the same interface. Each type can provide its own independent implementation of this interface.

About C#

- C# (pronounced "See Sharp") is a modern, object-oriented, and type-safe programming language.
- C# enables developers to build many types of secure and robust applications that run in .NET.
- C# has its roots in the C family of languages and will be immediately familiar to C, C++, Java, and JavaScript programmers.
- Several C# features such as Garbage collection Nullable types Exception handling asynchronous operations Lambda expressions help create robust and durable applications

Object-oriented programming in C#

- Encapsulation



main.cs

```
using System;
class Student
{
    private string name;
    private int age;

    public Student(string studentName, int studentAge)
    {
        name = studentName;
        age = studentAge;
    }
    public string GetName()
    {
        return name;
    }
    public void SetAge(int newAge)
    {
        if (newAge > 0)
        {
            age = newAge;
        }
        else
        {
            Console.WriteLine("Invalid age. Age must be a positive number.");
        }
    }
    public void DisplayInfo()
    {
        Console.WriteLine($"Student: {name}, Age: {age}");
    }
}
class Program
{
    static void Main()
    {
        Student student1 = new Student("Alice", 20);
        Console.WriteLine($"Original Age: {student1.GetName()} is {student1.GetName()}");
        student1.SetAge(21);
        Console.WriteLine($"Updated Age: {student1.GetName()} is {student1.GetName()}");
        student1.DisplayInfo();
        Console.ReadLine();
    }
}
```

Figure: Encapsulation

Object-oriented programming in C#

- Code in Figure-3 demonstrates the concept of encapsulation in C# In the *Student* class, *name* and *age* are private fields. This means they can't be accessed or modified directly from outside the class
- Only way to access them is via a getter method that is associated with th *Student* class

Object-oriented programming in C#

- Inheritance
- In the Figure-4 we can see Shape class is inherited by both Rectangle and Circle this is a example of Inheritance
- Abstraction
- In this code, *Shape* is an abstract class, which is a class that cannot be instantiated and is meant to be subclassed by other classes. It serves as a blueprint for its subclasses.
- The *Shape* class has an abstract method *Draw()*. Abstract methods are methods declared in an abstract class without any implementation. They must be overridden in any non-abstract class that directly inherits from the abstract class

Object-oriented programming in C#

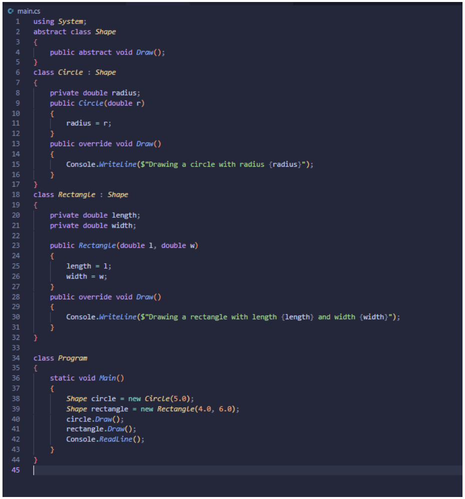
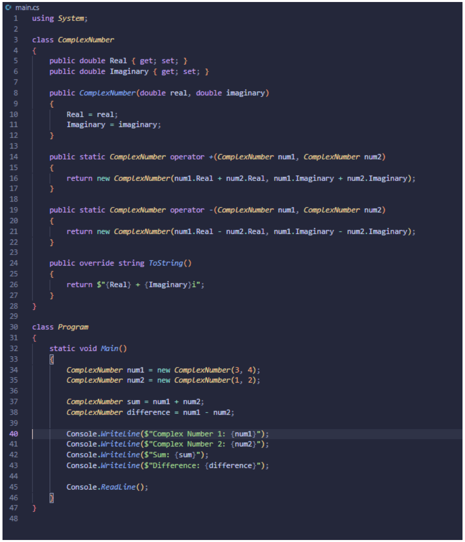

main.cs
```
using System;
abstract class Shape
{
    public abstract void Draw();
}
class Circle : Shape
{
    private double radius;
    public Circle(double r)
    {
        radius = r;
    }
    public override void Draw()
    {
        Console.WriteLine($"Drawing a circle with radius {radius}");
    }
}
class Rectangle : Shape
{
    private double length;
    private double width;

    public Rectangle(double l, double w)
    {
        length = l;
        width = w;
    }
    public override void Draw()
    {
        Console.WriteLine($"Drawing a rectangle with length {length} and width {width}");
    }
}

class Program
{
    static void Main()
    {
        Shape circle = new Circle(5.0);
        Shape rectangle = new Rectangle(4.0, 6.0);
        circle.Draw();
        rectangle.Draw();
        Console.ReadLine();
    }
}
```

Figure: Example for Abstraction

Object-oriented programming in C#

- Polymorphism
    - Fig-5 show **Operator Overloading** where + and - operator are overloaded
    - Fig-6 show **Method Overloading** where two Add function are defined with different parameters
    - Fig-7 shows **: Virtual/Method Overriding** where Virtual Function Draw is overridden Dynamically during run time

Object-oriented programming in C#

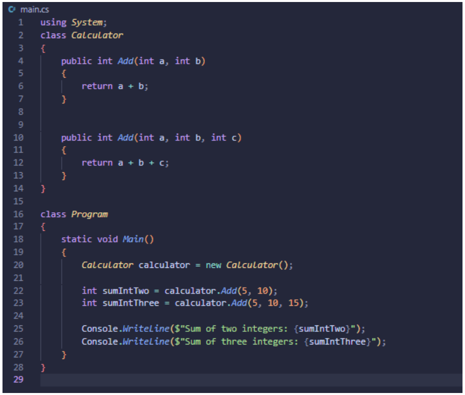


main.cs

```csharp
using System;

class ComplexNumber
{
    public double Real { get; set; }
    public double Imaginary { get; set; }

    public ComplexNumber(double real, double imaginary)
    {
        Real = real;
        Imaginary = imaginary;
    }

    public static ComplexNumber operator +(ComplexNumber num1, ComplexNumber num2)
    {
        return new ComplexNumber(num1.Real + num2.Real, num1.Imaginary + num2.Imaginary);
    }

    public static ComplexNumber operator -(ComplexNumber num1, ComplexNumber num2)
    {
        return new ComplexNumber(num1.Real - num2.Real, num1.Imaginary - num2.Imaginary);
    }

    public override string ToString()
    {
        return $"{Real} + {Imaginary}i";
    }
}

class Program
{
    static void Main()
    {
        ComplexNumber num1 = new ComplexNumber(3, 4);
        ComplexNumber num2 = new ComplexNumber(1, 2);

        ComplexNumber sum = num1 + num2;
        ComplexNumber difference = num1 - num2;

        Console.WriteLine($"Complex Number 1: {num1}");
        Console.WriteLine($"Complex Number 2: {num2}");
        Console.WriteLine($"Sum: {sum}");
        Console.WriteLine($"Difference: {difference}");

        Console.ReadLine();
    }
}
```

Figure: Operator Overloading

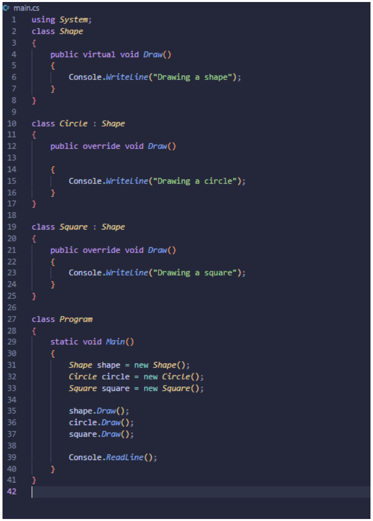
Object-oriented programming in C#


main.cs

```
using System;
class Calculator
{
    public int Add(int a, int b)
    {
        return a + b;
    }


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

class Program
{
    static void Main()
    {
        Calculator calculator = new Calculator();

        int sumIntTwo = calculator.Add(5, 10);
        int sumIntThree = calculator.Add(5, 10, 15);

        Console.WriteLine($"Sum of two integers: {sumIntTwo}");
        Console.WriteLine($"Sum of three integers: {sumIntThree}");
    }
}
```

Figure: method Overloading

Object-oriented programming in C#



main.cs

```
using System;
class Shape
{
    public virtual void Draw()
    {
        Console.WriteLine("Drawing a shape");
    }
}

class Circle : Shape
{
    public override void Draw()

    {
        Console.WriteLine("Drawing a circle");
    }
}

class Square : Shape
{
    public override void Draw()
    {
        Console.WriteLine("Drawing a square");
    }
}

class Program
{
    static void Main()
    {
        Shape shape = new Shape();
        Circle circle = new Circle();
        Square square = new Square();

        shape.Draw();
        circle.Draw();
        square.Draw();

        Console.ReadLine();
    }
}
```

Figure: Virtual/Method Overriding

Aspect-oriented programming

- Aspect oriented programming(AOP) as the name suggests uses aspects in programming. It can be defined as the breaking of code into different modules, also known as modularisation, where the aspect is the key unit of modularity
- In traditional programming, concerns such as logging, security, error handling, and other aspects are often intertwined with the core business logic of a program. As a result, modifying or extending one of these concerns can lead to changes in multiple places throughout the codebase.
- Aspect-oriented programming addresses this issue by providing a way to modularize cross-cutting concerns and encapsulate them in separate modules called aspects
- An aspect is a module that encapsulates a concern and defines how it should be woven into the main application.

Core Concepts of Aspect-oriented programming

- **Join point** A point during the execution of a program, such as the execution of a method or the handling of an exception
- **Advice** Action taken by an aspect at a particular join point. Different types of advice include "around", "before", and "after" advice.
  - **Before advice**: Advice that runs before a join point but that does not have the ability to prevent execution flow proceeding to the join point (unless it throws an exception).
  - **After advice**: Advice to be run after a join point completes normally
- **Pointcut** A set of join points. Pointcuts define a criterion for selecting join points where advice should be applied.
- **Aspect** A module containing advice and pointcut definitions. Aspects encapsulate cross-cutting concerns.
- **Weaving** linking aspects with other application types or objects to create an advised object.

About Spring

- Spring Framework is a comprehensive Java-based framework that simplifies and accelerates enterprise application development.
- It provides features such as dependency injection, aspect-oriented programming, and a wide range of modules for various functionalities
- Spring promotes modularity, testability, and ease of integration, making it a popular choice for building scalable and maintainable Java applications
- It also offers support for various architectural styles, including MVC for web applications. The framework simplifies configuration and promotes best practices in software design.

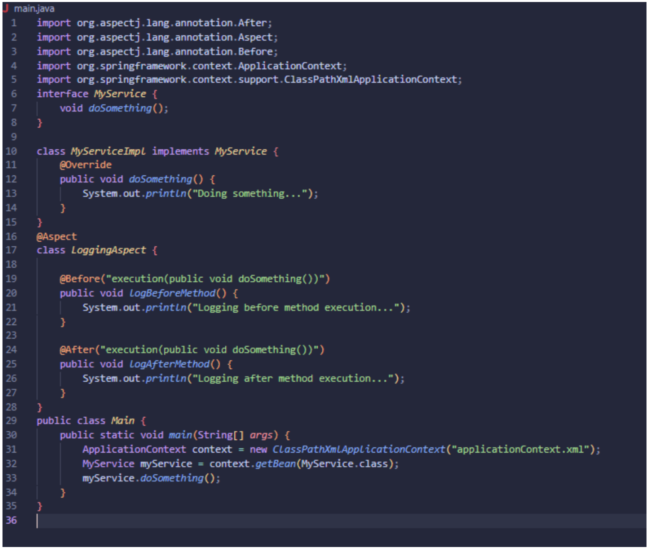
Aspect-oriented programming in Spring

- Join Point: A join point is a point in the execution of the program, such as the execution of a method or the handling of an exception. In Spring AOP, a join point always represents a method execution. In the provided code, the doSomething() method in the MyService interface is a join point.
- Advice: Advice is the action taken by an aspect at a particular join point. In the provided code, logBeforeMethod() and logAfterMethod() in the LoggingAspect class are advices. They are executed before and after the doSomething() method, respectively.
- Pointcut: A pointcut is a predicate that matches join points. Advice is associated with a pointcut expression and runs at any join point matched by the pointcut. In the provided code, the pointcut is defined as execution(public void doSomething()), which matches the execution of the doSomething() method in any class.

Aspect-oriented programming in Spring

- Aspect: An aspect is a modularization of a concern that cuts across multiple classes. It encapsulates behaviors that affect multiple classes into reusable modules. In the provided code, LoggingAspect is an aspect that encapsulates the cross-cutting concern of logging.
- Weaving: Weaving is the process of applying aspects to target objects to create new proxy objects. Spring AOP, like other pure Java AOP frameworks, performs weaving at runtime. In the provided code, the weaving is done when the MyService bean is retrieved from the application context and its doSomething() method is called.
- it is worth noteing that a config xml file with beans defined is needed to Spring to work

Aspect-oriented programming in Spring



main.java

```java
import org.aspectj.lang.annotation.After;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
interface MyService {
    void doSomething();
}

class MyServiceImpl implements MyService {
    @Override
    public void doSomething() {
        System.out.println("Doing something...");
    }
}
@Aspect
class LoggingAspect {

    @Before("execution(public void doSomething())")
    public void logBeforeMethod() {
        System.out.println("Logging before method execution...");
    }

    @After("execution(public void doSomething())")
    public void logAfterMethod() {
        System.out.println("Logging after method execution...");
    }
}
public class Main {
    public static void main(String[] args) {
        ApplicationContext context = new ClassPathXmlApplicationContext("applicationContext.xml");
        MyService myService = context.getBean(MyService.class);
        myService.doSomething();
    }
}
```

Figure: Spring Aspect Code example

Comparison

- Core Concept
  - AOP - Separates cross-cutting concerns from the main business logic by introducing aspects.
  - OOP - Organizes code around objects, encapsulating data and behavior.
- Main Idea
  - AOP - Provides a modular approach for concerns like logging, security, etc., enabling cleaner code structure.
  - OOP - Focuses on modeling real-world entities and their interactions.
- Core Concepts
  - AOP - Aspects, Join Points, Advice, Pointcuts, Weaving.
  - OOP - Classes, Objects, Inheritance, Encapsulation, Polymorphism.
- Use Cases
  - AOP - Useful for addressing concerns that cut across multiple modules, like logging, security, and transactions.
  - OOP - Suitable for modeling complex systems, promoting reusability and maintainability.

Discussions

When do we need them ?

- Object-Oriented Programming
  - When building systems that model real-world entities, their relationships, and behavior.
  - When there is a requirement for reusable code components that can be easily extended and customized.
  - When emphasizing data encapsulation, abstraction, and managing the complexity of large-scale systems.
- Aspect-Oriented Programming
  - When dealing with concerns that cut across multiple modules or layers, such as logging, security, and transaction management.
  - When there is a need to separate infrastructure-related code from business logic, maintaining a modular and scalable codebase.
  - When there is a requirement for consistent functionality across different parts of the application without duplicating code.

References

- www.codecademy.com/resources/docs/general/programming-paradigms/object-oriented-programming
- https://learn.microsoft.com/en-us/dotnet/csharp/
- https://www.c-sharpcorner.com/UploadFile/ff2f08/understanding-polymorphism-in-C-Sharp/
- https://www.c-sharpcorner.com/UploadFile/4624e9/abstraction-in-C-Sharp/
- https://unity.com/how-to#c-programming-unity
- https://www.linkedin.com/pulse/exploring-pros-cons-c-comprehensive-guide-baraa-mayasa/
- https://docs.spring.io/spring-framework/reference/core/aop.html
- https://chat.openai.com