МДК 01.04 Системное программирование

Лекция: Объектно-ориентированное программирование (ООП) в .NET

План лекции:

1. **Введение в ООП**
    - Что такое ООП и зачем оно нужно
    - Основные концепции и терминология
    - Преимущества ООП перед процедурным программированием
2. **Классы и объекты**
    - Определение классов
    - Создание объектов (экземпляров классов)
    - Конструкторы и деструкторы
    - Поля и методы
3. **Три столпа ООП**
    - Инкапсуляция
    - Наследование
    - Полиморфизм
4. **Дополнительные концепции ООП в C#**
    - Абстракция
    - Интерфейсы
    - Абстрактные классы
    - Модификаторы доступа
5. **Практические примеры и применение**

---

1. Введение в ООП

Что такое ООП? Объектно-ориентированное программирование (ООП) — это парадигма программирования, в которой программа организуется как совокупность взаимодействующих объектов. Каждый объект представляет собой экземпляр определенного класса, который выступает в роли "шаблона" или "чертежа".

Основные концепции:

- **Класс** — шаблон для создания объектов
- **Объект** — конкретный экземпляр класса
- **Атрибуты** — данные объекта (поля, свойства)
- **Методы** — функции, которые может выполнять объект

Преимущества ООП:

- Повторное использование кода
- Упрощение сопровождения и модификации
- Более четкая структура программы
- Упрощение работы в команде

Пример перехода от процедурного к ООП:

```
// Процедурный подход
string studentName = "Иван Иванов";
int studentAge = 20;
double studentGPA = 4.5;

void PrintStudentInfo(string name, int age, double gpa)
{
    Console.WriteLine($"Студент: {name}, Возраст: {age}, GPA: {gpa}");
}

// ООП подход
public class Student
{
    public string Name { get; set; }
    public int Age { get; set; }
    public double GPA { get; set; }

    public void PrintInfo()
    {
        Console.WriteLine($"Студент: {Name}, Возраст: {Age}, GPA: {GPA}");
    }
}

// Использование
Student student = new Student { Name = "Иван Иванов", Age = 20, GPA = 4.5 };
student.PrintInfo();
```

2. Классы и объекты

Определение классов: Класс — это пользовательский тип данных, который содержит данные (поля) и поведение (методы).

```
public class Car
{
    // Поля (данные)
    private string brand;
    private string model;
    private int year;
    private double speed;

    // Конструктор
```

```
    public Car(string brand, string model, int year)
    {
        this.brand = brand;
        this.model = model;
        this.year = year;
        this.speed = 0;
    }

    // Методы (поведение)
    public void Accelerate(double acceleration)
    {
        speed += acceleration;
        Console.WriteLine($"Скорость увеличена до {speed} км/ч");
    }

    public void Brake(double deceleration)
    {
        speed = Math.Max(0, speed - deceleration);
        Console.WriteLine($"Скорость уменьшена до {speed} км/ч");
    }

    public void DisplayInfo()
    {
        Console.WriteLine($"Автомобиль: {brand} {model} {year} года");
    }
}
```

Создание объектов:

```
// Создание объектов (экземпляров класса)
Car myCar = new Car("Toyota", "Camry", 2022);
Car friendsCar = new Car("BMW", "X5", 2023);

// Использование методов объектов
myCar.DisplayInfo();
myCar.Accelerate(50);
myCar.Brake(20);

friendsCar.DisplayInfo();
friendsCar.Accelerate(70);
```

Конструкторы и деструкторы:

```
public class BankAccount
{
    private string accountNumber;
    private decimal balance;
    private DateTime createdDate;
```

```
    // Конструктор по умолчанию
    public BankAccount()
    {
        accountNumber = GenerateAccountNumber();
        balance = 0;
        createdDate = DateTime.Now;
    }

    // Параметризованный конструктор
    public BankAccount(string accountNumber, decimal initialBalance)
    {
        this.accountNumber = accountNumber;
        this.balance = initialBalance;
        this.createdDate = DateTime.Now;
    }

    // Статический конструктор (вызывается один раз при первом обращении к классу)
    static BankAccount()
    {
        Console.WriteLine("Банковская система инициализирована");
    }

    // Деструктор (вызывается сборщиком мусора)
    ~BankAccount()
    {
        Console.WriteLine($"Счет {accountNumber} закрыт");
    }

    private string GenerateAccountNumber()
    {
        return "ACC" + DateTime.Now.Ticks.ToString().Substring(10);
    }

    public void Deposit(decimal amount)
    {
        balance += amount;
        Console.WriteLine($"Внесено {amount}. Баланс: {balance}");
    }

    public void Withdraw(decimal amount)
    {
        if (amount <= balance)
        {
            balance -= amount;
            Console.WriteLine($"Снято {amount}. Баланс: {balance}");
        }
        else
        {
            Console.WriteLine("Недостаточно средств");
        }
    }
}
```

3. Три столпа ООП

Инкапсуляция

Инкапсуляция — это механизм сокрытия внутренней реализации объекта и предоставления контролируемого интерфейса для работы с ним.

```
public class TemperatureSensor
{
    // Private поле - внутренняя реализация
    private double currentTemperature;
    private bool isActive;

    // Public свойства - контролируемый интерфейс
    public double CurrentTemperature
    {
        get { return currentTemperature; }
        private set
        {
            if (value >= -273.15) // Абсолютный ноль
                currentTemperature = value;
            else
                throw new ArgumentException("Температура не может быть ниже
абсолютного нуля");
        }
    }

    public bool IsActive
    {
        get { return isActive; }
        set
        {
            isActive = value;
            if (isActive)
                StartMonitoring();
            else
                StopMonitoring();
        }
    }

    // Public методы для взаимодействия
    public void UpdateTemperature(double newTemperature)
    {
        if (IsActive)
        {
            CurrentTemperature = newTemperature;
            CheckTemperatureThreshold();
        }
    }

    // Private методы - внутренняя реализация
    private void StartMonitoring()
```

```
    {
        Console.WriteLine("Мониторинг температуры запущен");
    }

    private void StopMonitoring()
    {
        Console.WriteLine("Мониторинг температуры остановлен");
    }

    private void CheckTemperatureThreshold()
    {
        if (CurrentTemperature > 100)
            Console.WriteLine("ВНИМАНИЕ: Температура превысила 100 градусов!");
    }
}

// Использование
TemperatureSensor sensor = new TemperatureSensor();
sensor.IsActive = true;
sensor.UpdateTemperature(25.5);
// sensor.CurrentTemperature = 30; // Ошибка! setter private
```

Наследование

Наследование — это механизм создания нового класса на основе существующего, с возможностью использования и расширения его функциональности.

```
// Базовый класс
public class Animal
{
    public string Name { get; set; }
    public int Age { get; set; }

    public Animal(string name, int age)
    {
        Name = name;
        Age = age;
    }

    public virtual void MakeSound()
    {
        Console.WriteLine("Животное издает звук");
    }

    public void Sleep()
    {
        Console.WriteLine($"{Name} спит");
    }
}

// Производные классы
```

```
public class Dog : Animal
{
    public string Breed { get; set; }

    public Dog(string name, int age, string breed) : base(name, age)
    {
        Breed = breed;
    }

    // Переопределение метода
    public override void MakeSound()
    {
        Console.WriteLine($"{Name} гавкает");
    }

    // Новый метод
    public void Fetch()
    {
        Console.WriteLine($"{Name} приносит палку");
    }
}

public class Cat : Animal
{
    public bool IsIndoor { get; set; }

    public Cat(string name, int age, bool isIndoor) : base(name, age)
    {
        IsIndoor = isIndoor;
    }

    public override void MakeSound()
    {
        Console.WriteLine($"{Name} мяукает");
    }

    public void ClimbTree()
    {
        Console.WriteLine($"{Name} лазает по деревьям");
    }
}

// Многоуровневое наследование
public class GermanShepherd : Dog
{
    public bool IsTrained { get; set; }

    public GermanShepherd(string name, int age, bool isTrained)
        : base(name, age, "Немецкая овчарка")
    {
        IsTrained = isTrained;
    }

    public void Guard()
```

```
    {
        Console.WriteLine($"{Name} охраняет территорию");
    }
}

// Использование
Animal genericAnimal = new Animal("Животное", 1);
Dog myDog = new Dog("Барсик", 3, "Дворняжка");
Cat myCat = new Cat("Мурка", 2, true);
GermanShepherd guardDog = new GermanShepherd("Рекс", 4, true);

genericAnimal.MakeSound(); // Животное издает звук
myDog.MakeSound();         // Барсик гавкает
myCat.MakeSound();         // Мурка мяукает

myDog.Sleep();             // Барсик спит
myDog.Fetch();             // Барсик приносит палку
guardDog.Guard();          // Рекс охраняет территорию
```

Полиморфизм

Полиморфизм — это возможность объектов разных классов обрабатываться как объекты общего базового класса, но при этом вызывать свои специфические методы.

```
public class Shape
{
    public virtual void Draw()
    {
        Console.WriteLine("Рисование фигуры");
    }

    public virtual double CalculateArea()
    {
        return 0;
    }
}

public class Circle : Shape
{
    public double Radius { get; set; }

    public Circle(double radius)
    {
        Radius = radius;
    }

    public override void Draw()
    {
        Console.WriteLine($"Рисование круга с радиусом {Radius}");
    }
```

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

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

    public Rectangle(double width, double height)
    {
        Width = width;
        Height = height;
    }

    public override void Draw()
    {
        Console.WriteLine($"Рисование прямоугольника {Width}x{Height}");
    }

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

public class Triangle : Shape
{
    public double Base { get; set; }
    public double Height { get; set; }

    public Triangle(double @base, double height)
    {
        Base = @base;
        Height = height;
    }

    public override void Draw()
    {
        Console.WriteLine($"Рисование треугольника с основанием {Base} и высотой
{Height}");
    }

    public override double CalculateArea()
    {
        return 0.5 * Base * Height;
    }
}

// Демонстрация полиморфизма
public class DrawingApp
{
```

```
    public void DrawShapes(List<Shape> shapes)
    {
        foreach (Shape shape in shapes)
        {
            shape.Draw(); // Полиморфный вызов
            double area = shape.CalculateArea();
            Console.WriteLine($"Площадь: {area:F2}");
            Console.WriteLine();
        }
    }
}

// Использование
List<Shape> shapes = new List<Shape>
{
    new Circle(5),
    new Rectangle(4, 6),
    new Triangle(3, 4),
    new Circle(3)
};

DrawingApp app = new DrawingApp();
app.DrawShapes(shapes);
```

Сравнение new и override

```
public class BaseClass
{
    public void NormalMethod()
    {
        Console.WriteLine("BaseClass.NormalMethod");
    }

    public virtual void VirtualMethod()
    {
        Console.WriteLine("BaseClass.VirtualMethod");
    }
}

public class DerivedClass : BaseClass
{
    // Сокрытие метода (new)
    public new void NormalMethod()
    {
        Console.WriteLine("DerivedClass.NormalMethod (new)");
    }

    // Переопределение метода (override)
    public override void VirtualMethod()
    {
        Console.WriteLine("DerivedClass.VirtualMethod (override)");
```

```
    }
}

// Тестирование
class TestProgram
{
    static void Main()
    {
        BaseClass baseObj = new BaseClass();
        DerivedClass derivedObj = new DerivedClass();
        BaseClass polyObj = new DerivedClass(); // Полиморфизм

        Console.WriteLine("=== BaseClass ===");
        baseObj.NormalMethod();     // BaseClass.NormalMethod
        baseObj.VirtualMethod();    // BaseClass.VirtualMethod

        Console.WriteLine("\n=== DerivedClass ===");
        derivedObj.NormalMethod();  // DerivedClass.NormalMethod (new)
        derivedObj.VirtualMethod(); // DerivedClass.VirtualMethod (override)

        Console.WriteLine("\n=== Полиморфизм (BaseClass = new DerivedClass())
===");
        polyObj.NormalMethod();     // BaseClass.NormalMethod (new не работает
полиморфно)
        polyObj.VirtualMethod();    // DerivedClass.VirtualMethod (override
работает полиморфно)
    }
}
```

Ключевое слово `new`:

- Создание объектов - основной способ инстанцирования классов
- Сокрытие членов - позволяет скрыть члены базового класса без полиморфного поведения
- Альтернатива `override` - когда полиморфизм не нужен или нежелателен
- Требует осторожности - может привести к неожиданному поведению если используется неправильно

4. Дополнительные концепции ООП в C#

Абстракция

Абстракция — это процесс выделения основных характеристик объекта и игнорирования несущественных деталей.

```
// Абстрактный класс
public abstract class Vehicle
{
    public string Manufacturer { get; set; }
```

```
    public string Model { get; set; }
    public int Year { get; set; }

    // Абстрактный метод - без реализации
    public abstract void Start();
    public abstract void Stop();

    // Виртуальный метод - с реализацией по умолчанию
    public virtual void DisplayInfo()
    {
        Console.WriteLine($"Транспортное средство: {Manufacturer} {Model} {Year}
года");
    }
}

public class Car : Vehicle
{
    public int NumberOfDoors { get; set; }

    public override void Start()
    {
        Console.WriteLine("Заводим двигатель автомобиля");
        Console.WriteLine("Проверяем ремни безопасности");
        Console.WriteLine("Автомобиль готов к движению");
    }

    public override void Stop()
    {
        Console.WriteLine("Нажимаем на тормоз");
        Console.WriteLine("Переключаем передачу в паркинг");
        Console.WriteLine("Заглушаем двигатель");
    }

    public override void DisplayInfo()
    {
        base.DisplayInfo();
        Console.WriteLine($"Количество дверей: {NumberOfDoors}");
    }
}

public class Motorcycle : Vehicle
{
    public bool HasSideCar { get; set; }

    public override void Start()
    {
        Console.WriteLine("Заводим двигатель мотоцикла");
        Console.WriteLine("Надеваем шлем");
        Console.WriteLine("Мотоцикл готов к движению");
    }

    public override void Stop()
    {
        Console.WriteLine("Сбрасываем газ");
```

```
        Console.WriteLine("Нажимаем на тормоз");
        Console.WriteLine("Заглушаем двигатель");
    }
}
```

Интерфейсы

Интерфейсы определяют контракт, который должны реализовать классы.

```
// Интерфейсы
public interface ILogger
{
    void Log(string message);
    void LogError(string error);
}

public interface IDatabaseOperations
{
    void Connect();
    void Disconnect();
    void ExecuteQuery(string query);
}

public interface IAuthenticable
{
    bool Login(string username, string password);
    void Logout();
}

// Класс, реализующий несколько интерфейсов
public class DatabaseManager : ILogger, IDatabaseOperations, IAuthenticable
{
    private bool isConnected = false;
    private bool isAuthenticated = false;

    // Реализация ILogger
    public void Log(string message)
    {
        Console.WriteLine($"[INFO] {DateTime.Now}: {message}");
    }

    public void LogError(string error)
    {
        Console.WriteLine($"[ERROR] {DateTime.Now}: {error}");
    }

    // Реализация IDatabaseOperations
    public void Connect()
    {
        if (!isAuthenticated)
        {
```

```
            LogError("Попытка подключения без аутентификации");
            throw new InvalidOperationException("Требуется аутентификация");
        }

        isConnected = true;
        Log("Подключение к базе данных установлено");
    }

    public void Disconnect()
    {
        isConnected = false;
        Log("Отключение от базы данных");
    }

    public void ExecuteQuery(string query)
    {
        if (!isConnected)
        {
            LogError("Попытка выполнения запроса без подключения");
            throw new InvalidOperationException("Требуется подключение к БД");
        }

        Log($"Выполнение запроса: {query}");
        // Логика выполнения запроса
    }

    // Реализация IAuthenticable
    public bool Login(string username, string password)
    {
        // Упрощенная логика аутентификации
        if (username == "admin" && password == "password")
        {
            isAuthenticated = true;
            Log($"Пользователь {username} успешно аутентифицирован");
            return true;
        }

        LogError($"Неудачная попытка аутентификации для пользователя {username}");
        return false;
    }

    public void Logout()
    {
        isAuthenticated = false;
        isConnected = false;
        Log("Пользователь вышел из системы");
    }
}

// Другой класс, реализующий тот же интерфейс
public class FileLogger : ILogger
{
    private string filePath;
```

```
    public FileLogger(string path)
    {
        filePath = path;
    }

    public void Log(string message)
    {
        File.AppendAllText(filePath, $"[INFO] {DateTime.Now}: {message}\n");
    }

    public void LogError(string error)
    {
        File.AppendAllText(filePath, $"[ERROR] {DateTime.Now}: {error}\n");
    }
}
```

Модификаторы доступа

```
public class AccessModifiersDemo
{
    // Public - доступен отовсюду
    public string PublicField = "Public";

    // Private - доступен только внутри класса
    private string PrivateField = "Private";

    // Protected - доступен внутри класса и производных классов
    protected string ProtectedField = "Protected";

    // Internal - доступен в пределах сборки
    internal string InternalField = "Internal";

    // Protected Internal - доступен в пределах сборки и производных классах
    protected internal string ProtectedInternalField = "Protected Internal";

    // Private Protected - доступен в пределах сборки только производным классам
    private protected string PrivateProtectedField = "Private Protected";

    public void DemonstrateAccess()
    {
        // Все поля доступны внутри класса
        Console.WriteLine(PublicField);
        Console.WriteLine(PrivateField);
        Console.WriteLine(ProtectedField);
        Console.WriteLine(InternalField);
        Console.WriteLine(ProtectedInternalField);
        Console.WriteLine(PrivateProtectedField);
    }
}

public class DerivedClass : AccessModifiersDemo
```

```
{
    public void DemonstrateInheritedAccess()
    {
        // Доступны public, protected, protected internal, private protected
        Console.WriteLine(PublicField);
        // Console.WriteLine(PrivateField); // Ошибка - private недоступен
        Console.WriteLine(ProtectedField);
        Console.WriteLine(InternalField); // Доступен, если в той же сборке
        Console.WriteLine(ProtectedInternalField);
        Console.WriteLine(PrivateProtectedField); // Доступен, если в той же
сборке
    }
}
```

5. Практические примеры и применение

Реальный пример: система управления библиотекой

```
// Базовые интерфейсы
public interface ILibraryItem
{
    string Title { get; }
    string Author { get; }
    int Year { get; }
    bool IsAvailable { get; }
    void CheckOut();
    void Return();
}

public interface ISearchable
{
    bool MatchesSearch(string searchTerm);
}

// Абстрактный базовый класс
public abstract class LibraryItem : ILibraryItem, ISearchable
{
    public string Title { get; protected set; }
    public string Author { get; protected set; }
    public int Year { get; protected set; }
    public bool IsAvailable { get; protected set; } = true;

    protected LibraryItem(string title, string author, int year)
    {
        Title = title;
        Author = author;
        Year = year;
    }

    public virtual void CheckOut()
    {
```

```
        if (IsAvailable)
        {
            IsAvailable = false;
            Console.WriteLine($"'{Title}' выдана");
        }
        else
        {
            Console.WriteLine($"'{Title}' уже выдана");
        }
    }

    public virtual void Return()
    {
        IsAvailable = true;
        Console.WriteLine($"'{Title}' возвращена");
    }

    public virtual bool MatchesSearch(string searchTerm)
    {
        return Title.Contains(searchTerm, StringComparison.OrdinalIgnoreCase) ||
               Author.Contains(searchTerm, StringComparison.OrdinalIgnoreCase);
    }

    public abstract void DisplayInfo();
}

// Конкретные классы
public class Book : LibraryItem
{
    public string ISBN { get; private set; }
    public int PageCount { get; private set; }

    public Book(string title, string author, int year, string isbn, int pageCount)
        : base(title, author, year)
    {
        ISBN = isbn;
        PageCount = pageCount;
    }

    public override void DisplayInfo()
    {
        Console.WriteLine($"КНИГА: {Title}");
        Console.WriteLine($"Автор: {Author}");
        Console.WriteLine($"Год: {Year}");
        Console.WriteLine($"ISBN: {ISBN}");
        Console.WriteLine($"Страниц: {PageCount}");
        Console.WriteLine($"Статус: {(IsAvailable ? "Доступна" : "Выдана")}");
        Console.WriteLine();
    }
}

public class Magazine : LibraryItem
{
    public int IssueNumber { get; private set; }
```

```
    public string Publisher { get; private set; }

    public Magazine(string title, string publisher, int year, int issueNumber)
        : base(title, publisher, year)
    {
        Publisher = publisher;
        IssueNumber = issueNumber;
    }

    public override void DisplayInfo()
    {
        Console.WriteLine($"ЖУРНАЛ: {Title}");
        Console.WriteLine($"Издатель: {Publisher}");
        Console.WriteLine($"Год: {Year}");
        Console.WriteLine($"Номер: {IssueNumber}");
        Console.WriteLine($"Статус: {(IsAvailable ? "Доступен" : "Выдан")}");
        Console.WriteLine();
    }
}

// Класс для управления библиотекой
public class Library
{
    private List<LibraryItem> items = new List<LibraryItem>();
    private List<LibraryMember> members = new List<LibraryMember>();

    public void AddItem(LibraryItem item)
    {
        items.Add(item);
        Console.WriteLine($"Добавлен: {item.Title}");
    }

    public void RegisterMember(LibraryMember member)
    {
        members.Add(member);
        Console.WriteLine($"Зарегистрирован читатель: {member.Name}");
    }

    public List<LibraryItem> Search(string searchTerm)
    {
        return items.Where(item => item.MatchesSearch(searchTerm)).ToList();
    }

    public void DisplayAllItems()
    {
        Console.WriteLine("=== ВСЕ МАТЕРИАЛЫ БИБЛИОТЕКИ ===");
        foreach (var item in items)
        {
            item.DisplayInfo();
        }
    }
}

public class LibraryMember
```

```
{
    public string Name { get; private set; }
    public string MemberId { get; private set; }
    private List<ILibraryItem> borrowedItems = new List<ILibraryItem>();

    public LibraryMember(string name, string memberId)
    {
        Name = name;
        MemberId = memberId;
    }

    public void BorrowItem(ILibraryItem item)
    {
        if (item.IsAvailable)
        {
            item.CheckOut();
            borrowedItems.Add(item);
            Console.WriteLine($"{Name} взял(а) '{item.Title}'");
        }
        else
        {
            Console.WriteLine($"'{item.Title}' недоступна для выдачи");
        }
    }

    public void ReturnItem(ILibraryItem item)
    {
        if (borrowedItems.Contains(item))
        {
            item.Return();
            borrowedItems.Remove(item);
            Console.WriteLine($"{Name} вернул(а) '{item.Title}'");
        }
        else
        {
            Console.WriteLine($"{Name} не брал(а) '{item.Title}'");
        }
    }

    public void DisplayBorrowedItems()
    {
        Console.WriteLine($"=== КНИГИ НА РУКАХ У {Name} ===");
        foreach (var item in borrowedItems)
        {
            Console.WriteLine($"- {item.Title} ({item.Author})");
        }
        Console.WriteLine();
    }
}

// Использование системы
class Program
{
    static void Main()
```

```
    {
        // Создание библиотеки
        Library library = new Library();

        // Добавление материалов
        library.AddItem(new Book("Война и мир", "Лев Толстой", 1869, "978-5-389-
00001-1", 1225));
        library.AddItem(new Book("Преступление и наказание", "Федор Достоевский",
1866, "978-5-389-00002-8", 672));
        library.AddItem(new Magazine("National Geographic", "National Geographic
Society", 2023, 5));

        // Регистрация читателей
        LibraryMember member1 = new LibraryMember("Иван Петров", "M001");
        LibraryMember member2 = new LibraryMember("Мария Сидорова", "M002");
        library.RegisterMember(member1);
        library.RegisterMember(member2);

        // Работа с библиотекой
        library.DisplayAllItems();

        // Поиск
        var searchResults = library.Search("Толстой");
        Console.WriteLine("Результаты поиска по 'Толстой':");
        foreach (var item in searchResults)
        {
            item.DisplayInfo();
        }

        // Выдача и возврат книг
        var book = library.Search("Война и мир").First() as Book;
        member1.BorrowItem(book);
        member1.DisplayBorrowedItems();

        member1.ReturnItem(book);
        member1.DisplayBorrowedItems();
    }
}
```

Резюме лекции:

Ключевые моменты ООП:

1. **Инкапсуляция** — сокрытие внутренней реализации и предоставление контролируемого интерфейса
2. **Наследование** — создание новых классов на основе существующих с возможностью расширения функциональности
3. **Полиморфизм** — возможность объектов разных классов обрабатываться единообразно через общий интерфейс

4. **Абстракция** — выделение существенных характеристик объекта и игнорирование несущественных деталей

Преимущества ООП в .NET:

- Упрощение разработки сложных систем
- Повторное использование кода через наследование и композицию
- Упрощение тестирования и отладки
- Лучшая организация кода и структуры проекта
- Поддержка современных практик разработки (SOLID, паттерны проектирования)

Рекомендации по применению:

- Используйте инкапсуляцию для защиты внутреннего состояния объектов
- Применяйте наследование для создания иерархий связанных объектов
- Используйте полиморфизм для создания гибких и расширяемых систем
- Отдавайте предпочтение композиции перед наследованием там, где это уместно
- Следуйте принципам SOLID для создания качественного кода

ООП является фундаментальной парадигмой в .NET разработке, и глубокое понимание ее принципов необходимо для создания эффективных, поддерживаемых и масштабируемых приложений.

Контрольные вопросы

- Что такое объектно-ориентированное программирование и каковы его основные преимущества?
- Чем отличается класс от объекта? Приведите примеры.
- Что такое конструктор класса? Какие типы конструкторов вы знаете?
- Что такое поля и методы класса? Как они связаны с состоянием и поведением объекта?
- Какие модификаторы доступа существуют в C# и чем они отличаются? Что такое инкапсуляция? Приведите пример из реальной жизни и реализуйте его в коде.
- Объясните принцип наследования. Какие преимущества он дает?
- Что такое полиморфизм? В чем разница между переопределением (override) и сокрытием (new)?
- Чем абстрактный класс отличается от интерфейса? Когда что использовать?
- Что такое виртуальные методы и для чего они нужны?