Тема 1. Основы ООП.

Цель занятия:

Изучить базовые концепции ООП, их реализацию на языке C#.

Учебные вопросы:

1. Введение.
2. Основные понятия ООП (Класс, Объект, Инкапсуляция, Наследование, Полиморфизм).
3. Поля и свойства.
4. Модификаторы доступа.
5. Методы.
6. Конструкторы.

1. Введение.

Объектно-ориентированное программирование (ООП) — это парадигма программирования, основанная на концепции объектов.

Объекты — это сущности, которые объединяют данные (поля) и логику работы с ними (методы).

ООП помогает моделировать программное обеспечение так, чтобы его структура напоминала реальные объекты и их взаимодействия.

Ключевые характеристики ООП:

Инкапсуляция: объединение данных и методов в одной сущности (объекте) и сокрытие деталей реализации.

Наследование: возможность создания нового класса на основе уже существующего, чтобы переиспользовать его функционал.

Полиморфизм: способность объектов разного типа реагировать на одинаковые вызовы методов.

Преимущества ООП:

Модульность: разделение программы на независимые компоненты.

Повторное использование кода: использование классов и методов в разных частях приложения.

Удобство разработки: понятность структуры кода и уменьшение количества ошибок.

Упрощение расширения: добавление новой функциональности без значительных изменений в уже написанном коде.

Роль ООП в современном программировании:

- Используется для создания крупных, сложных и долгосрочных проектов.
- Находит применение в большинстве областей разработки: от веб-приложений до сложных систем искусственного интеллекта.
- Подходит для командной работы, так как позволяет распределять задачи между разработчиками, фокусируясь на отдельных компонентах системы.

2. Основные понятия ООП.

Класс и объект — это фундаментальные понятия объектно-ориентированного программирования.

Класс — это абстрактный шаблон или чертёж, описывающий свойства (поля) и поведение (методы), которыми будут обладать создаваемые на его основе объекты. Пример: класс Автомобиль может описывать общие свойства, такие как марка, модель, и действия, такие как ехать или остановиться.

Объект — это конкретный экземпляр класса, созданный в памяти программы. Каждый объект имеет собственное состояние (значения свойств) и может выполнять действия, описанные в классе. Пример: объект класса Автомобиль — это, например, "Toyota Corolla 2022", которая едет со скоростью 60 км/ч.

Примеры из реальной жизни:

Класс: Человек. Поля: имя, возраст. Методы: говорить(), ходить().

Объект: конкретный человек — Иван, 25 лет, который может говорить и ходить.

Класс: Книга. Поля: название, автор, количество страниц. Методы: читать(), закрыть().

Объект: книга "Война и мир", автор — Лев Толстой, 1225 страниц.

Примеры

```
public class Car
{
    Ссылок: 1
    public string Brand { get; set; }
    Ссылок: 1
    public string Model { get; set; }


    Ссылок: 0
    public void Drive()
    {
        Console.WriteLine($"{Brand} {Model} is driving.");
    }
}
```

Примеры

```
// Создание объекта
Car myCar = new Car
{
    Brand = "Toyota",
    Model = "Corolla"
};
myCar.Drive();  // Output: Toyota Corolla is driving.
```

3. Поля и свойства.

Поле класса — это переменная, объявленная внутри класса, которая используется для хранения данных, связанных с этим классом или его экземплярами.

Поле описывает **состояние объекта** и доступно для методов, свойств и других членов класса.

Поля определяются внутри класса, но вне методов или свойств.

Поле может быть публичным (**public**), приватным (**private**), защищённым (**protected**) или доступным внутри сборки (**internal**).

Поле может быть статичным, т.е. общим для всех объектов класса

Пример объявления полей класса:

```
internal class Car
{
    // публичное поле класса
    public string type;

    // скрытое поле класса
    private string brand;

    // Статическое поле (общее для всех объектов класса)
    public static int CarCount;

}
```

Свойства класса

Свойства класса в C# — это специальные члены класса, которые позволяют управлять доступом к полям объекта.

Свойства предоставляют способ получения и установки значений полей с помощью методов **get** и **set**, сохраняя при этом синтаксис работы с полями.

При использовании **set** значение передаётся в параметр **value**.

Особенности свойств

- Инкапсуляция данных. Свойства обеспечивают контроль над доступом к полям, позволяя добавлять проверку или изменять логику без прямого обращения к полям.
- Если дополнительная логика в get/set не нужна, можно использовать сокращённый синтаксис.
- Модификаторы доступа. Вы можете задавать различные модификаторы для get и set.

```
public class BankAccount
{
    private decimal balance; // Скрытое поле

    // Свойство для управления доступом к балансу
    Ссылок: 6
    public decimal Balance
    {
        get { return balance; }  // Возврат значения
        set { balance = value; } // Установка значения
    }
```

```
BankAccount account = new BankAccount();
account.Balance = 10000;                  // Установка значения через set
Console.WriteLine(account.Balance); // Получение значения через get
```

Автоматические свойства.

Если в свойстве нет необходимости добавлять дополнительную логику, можно использовать автоматические свойства, которые автоматически создают скрытое поле для хранения данных.

```
Ссылок: 0
public class Car
{
    Ссылок: 0
    public string Brand { get; set; } // Автоматическое свойство
    // Компилятор автоматически создаст скрытое поле для хранения значения.
}
```

Реализация логики внутри get и set

В get и set можно добавлять логику для выполнения проверок или преобразований данных.

Пример: проверка значений в set.

```
public decimal Balance
{
    get { return balance; }
    set
    {
        if (value < 0)
            throw new ArgumentException("Баланс не может быть отрицательным.");
        balance = value;
    }
}
```

Вычисляемые свойства.

Свойства могут быть только с get (т.е. только для чтения). Например, свойство, которое возвращает вычисляемое значение:

```
public class Rectangle
{
    Ссылок: 2
    public double Width { get; set; }
    Ссылок: 2
    public double Height { get; set; }

    Ссылок: 0
    public double Area // Только для чтения
    {
        get { return Width * Height; }
    }
}
```

Пример: условное управление полем

```
public class Person
{
    private int age;
    Ссылок: 1
    public int Age
    {
        get { return age; }
        set
        {
            if (value < 0)
                throw new ArgumentException("Возраст не может быть отрицательным.");
            if (value > 150)
                throw new ArgumentException("Возраст не может превышать 150 лет.");
            age = value;
        }
    }
}
```

Свойства помогают:

1. **Инкапсулировать данные**. Пользователь взаимодействует со свойствами, а не с полями.
2. **Добавлять логику**. Проверка значений, вычисления или преобразования при доступе.
3. **Упрощать код**. Автоматические свойства обеспечивают компактный синтаксис.

4. Модификаторы доступа.

Модификаторы доступа определяют, где в программе можно использовать (видеть или изменять) элементы класса (поля, свойства, методы и даже сам класс).

1. public

Доступен из любой точки программы.

Элемент с модификатором public может быть использован:

Внутри того же класса.

В других классах (в том числе находящихся в другой сборке).

Обычно используется для свойств, методов или классов, которые должны быть доступны всем.

2. private

Доступен только внутри текущего класса.

Элемент с модификатором private не может быть использован за пределами класса, где он объявлен.

Обычно используется для полей и методов, которые скрыты от внешнего кода и используются только внутри класса.

3. protected

Доступен внутри текущего класса и производных классов.

Элемент с модификатором **protected** не доступен из вне, но может быть использован в классах, которые наследуются от текущего.

Обычно используется для методов или данных, которые должны быть доступны только в рамках семейства классов (класс + его наследники).

4. internal

Доступен внутри текущей сборки (assembly).

Элемент с модификатором **internal** виден только в коде, который находится в одной сборке (например, в одном .dll или .exe).

Используется, когда нужно ограничить доступ к элементу только внутри проекта.

5. protected internal

Доступен внутри текущей сборки или в производных классах.

Комбинация **protected** и **internal** означает, что элемент доступен:

Либо в коде текущей сборки.

Либо в коде производных классов, даже если они находятся в другой сборке.

Модификатор	Доступ внутри класса	Доступ в наследниках	Доступ в сборке	Доступ из другой сборки
`public`	✔	✔	✔	✔
`private`	✔	✘	✘	✘
`protected`	✔	✔	✘	✘
`internal`	✔	✔	✔	✘
`protected internal`	✔	✔	✔	✔ (только в наследниках)

5. Методы класса

Методы класса в C# — это функции, которые определены внутри класса и описывают действия, которые может выполнять объект данного класса.

Они работают с данными класса (поля, свойства) или выполняют другие задачи.

Основные характеристики методов

- Методы определяются внутри класса и являются его членами.
- Методы могут быть: public (доступны везде), private (доступны только внутри класса), protected (доступны внутри класса и его наследников), internal (доступны внутри одной сборки).
- Методы могут принимать параметры и возвращать результат.

Типы методов:

- Экземплярные (работают с конкретным объектом класса)
- Статические (принадлежат самому классу, вызываются без создания объекта).

Синтаксис метода:

```
[модификатор доступа] [возвращаемый тип] ИмяМетода([параметры])
{
    // Тело метода
    return [значение]; // (если требуется)
}
```

Пример простого метода:

```
Ссылок: 0
public class Calculator
{
    Ссылок: 0
    public int Add(int a, int b) // Метод принимает два числа и возвращает их сумму
    {
        return a + b;
    }
}
```

```
Ссылок: 0
static void Main()
{
    Calculator calculator = new Calculator();
    int result = calculator.Add(3, 5); // Вызов метода
    Console.WriteLine($"Результат: {result}"); // Результат: 8
}
```

Метод может ничего не принимать и не возвращать:

```
Ссылок: 0
public class Greeter
{
    Ссылок: 0
    public void SayHello()
    {
        Console.WriteLine("Привет, мир!");
    }
}
```

```
Ссылок: 0
static void Main()
{
    Greeter greeter = new Greeter();
    greeter.SayHello(); // Вывод: Привет, мир!
}
```

Метод с параметрами по умолчанию:

```
Ссылок: 0
public class Greeter
{
    Ссылок: 0
    public void Greet(string name = "Гость")
    {
        Console.WriteLine($"Привет, {name}!");
    }

}
```

```
Ссылок: 0
static void Main()
{
    Greeter greeter = new Greeter();
    greeter.Greet();              // Привет, Гость!
    greeter.Greet("Алексей");     // Привет, Алексей!
}
```

Статические методы принадлежат классу и вызываются без создания объекта:

```
Ссылок: 0
public class MathUtils
{
    Ссылок: 0
    public static int Multiply(int a, int b)
    {
        return a * b;
    }
}
```

```
Ссылок: 0
static void Main()
{
    int result = MathUtils.Multiply(4, 5); // Вызов статического метода
    Console.WriteLine($"Результат: {result}"); // Результат: 20
}
```

Методы могут иметь одно и то же имя, но разные параметры (**перегрузка**):

```
Ссылок: 0
public class Printer
{
    Ссылок: 0
    public void Print(string message)
    {
        Console.WriteLine($"Сообщение: {message}");
    }

    Ссылок: 0
    public void Print(int number)
    {
        Console.WriteLine($"Число: {number}");
    }
}
```

```
Ссылок: 0
public class Calculator
{
    Ссылок: 0
    public double Summa(double a, double b)
    {
        return a + b;
    }

    Ссылок: 0
    public int Summa(int a, int b)
    {
        return a + b;
    }

    Ссылок: 0
    public int Summa(int a, int b, int c) => a + b + c;
}
```

Перегрузка методов в C# — это способность класса иметь несколько методов с одним и тем же именем, но с разными параметрами. Это позволяет вызывать метод в зависимости от переданных аргументов.

Основные характеристики:

- Методы с одним именем, но разной сигнатурой, т.е. различия могут быть в типе, количестве или порядке параметров.
- Возвращаемый тип не учитывается при перегрузке.

Методы-расширения (Extension Methods) в C# — это способ добавлять новые методы в существующие классы или интерфейсы **без их изменения** или создания новых наследников.

Они полезны для улучшения функциональности уже существующих типов, когда у вас нет доступа к их коду (например, классы стандартных библиотек).

Основные характеристики:

- Метод-расширение — это статический метод, который выглядит как обычный метод класса, но вызывается для объекта.
- Первый параметр метода-расширения должен содержать ключевое слово **this** и ссылаться на тип, который вы расширяете.
- Методы-расширения обычно определяются в статическом классе.

Пример расширения для стандартного класса:

```
Ссылок: 0
public static class StringExtension
{
    // Метод-расширение для стандартного класса string
    Ссылок: 1
    public static string ToCapitalized(this string str)
    {
        if (string.IsNullOrEmpty(str)) return str;
        return char.ToUpper(str[0]) + str.Substring(1).ToLower();
    }
}
```

```
Ссылок: 0
static void Main(string[] args)
{
    // Использование
    string text = "hello world";
    string capitalized = text.ToCapitalized();
    Console.WriteLine(capitalized); // Вывод: "Hello world"
}
```

Пример расширения для пользовательского класса:

```
public class Point  // пользовательский класс
{
    public int X { get; set; }
    public int Y { get; set; }
}

// класс, в котором определён метод-расширение
public static class PointExtension
{
    // метод-расширение для типа Point
    public static double DistanceToOrigin(this Point point)
    {
        return Math.Sqrt(point.X * point.X + point.Y * point.Y);
    }
}
```

```
Ссылок: 0
static void Main(string[] args)
{
    // Использование
    Point p = new Point { X = 3, Y = 4 };
    Console.WriteLine(p.DistanceToOrigin()); // Вывод: 5
}
```

6. Конструкторы

Конструктор — это специальный метод в классе, который вызывается автоматически при создании объекта.

Конструктор используется для инициализации полей или выполнения начальных действий, необходимых при создании экземпляра класса.

Ключевые особенности конструктора:

- Имя конструктора совпадает с именем класса.
- Конструктор не имеет возвращаемого значения, даже void.
- Конструкторы могут быть перегружены. В классе может быть несколько конструкторов с разными наборами параметров.
- Если конструктор не задан явно, компилятор создаёт конструктор по умолчанию.
- Могут быть вызваны как другие методы, так и перегруженные версии через ключевое слово this.

Типы конструкторов в C#:

1. Конструктор по умолчанию

Если вы не создадите конструктор явно, компилятор предоставит его автоматически. Такой конструктор не принимает аргументов.

Если вы объявите хотя бы один конструктор, конструктор по умолчанию перестаёт создаваться автоматически.

Пример конструктора по умолчанию

```
class Person
{
    public string Name;
    public int Age;

}
```

```
Person person = new Person(); // Вызов конструктора по умолчанию
Console.WriteLine($"{person.Name}, {person.Age}"); // null, 0
```

2. Пользовательский конструктор

Конструктор, который вы явно определяете в классе для инициализации полей объекта.

```
// Пользовательский конструктор
Ссылок: 0
public Person(string name, int age)
{
    Name = name;
    Age = age;
}
```

3. Перегрузка конструкторов

В C# можно определить несколько конструкторов с разными параметрами.

```
// Конструктор с двумя параметрами
Ссылок: 0
public Person(string name, int age)
{
    Name = name;
    Age = age;
}
// Конструктор с одним параметром
Ссылок: 0
public Person(string name)
{
    Name = name;
    Age = 0; // Возраст по умолчанию
}
```

4. Конструктор с параметрами по умолчанию

```
class Person
{
    public string Name;
    public int Age;
    // Конструктор с параметрами по умолчанию
    Ссылок: 0
    public Person(string name = "Неизвестно", int age = 0)
    {
        Name = name;
        Age = age;
    }
}
```

5. Конструкторы с использованием this

Ключевое слово **this** позволяет вызывать другой конструктор внутри текущего.

```
// Конструктор с двумя параметрами
Ссылок: 2
public Person(string name, int age)
{
    Name = name;
    Age = age;
}
// Конструктор, который вызывает другой
Ссылок: 0
public Person(string name) : this(name, 0) { }
// Конструктор по умолчанию
Ссылок: 1
public Person() : this("Неизвестно", 0) { }
```

Ключевое слово **this** в C# используется для ссылки на текущий экземпляр класса или структуры. Оно помогает разграничить контекст текущего объекта от других контекстов (например, от локальных переменных или параметров метода).

this используется, чтобы явно указать, что вы работаете с полями или методами текущего объекта. Это полезно, когда локальная переменная или параметр метода имеют то же имя, что и поле объекта.

С помощью **this** можно вызвать один конструктор класса из другого. Это называется цепочкой вызова конструкторов

пример: Банковский счет.

```
public class BankAccount
{
    // Приватные поля
    private string accountNumber;  // Номер счёта
    private string accountHolder;  // Владелец счёта
    private decimal balance;       // Баланс

    // Свойства для доступа к полям
    Ссылок: 2
    public string AccountNumber
    {
        get { return accountNumber; } // Только для чтения
        // Устанавливается только внутри класса
        private set { accountNumber = value; }
    }

    Ссылок: 2
    public string AccountHolder
    {
        get { return accountHolder; } // Только для чтения
        // Устанавливается только внутри класса
        private set { accountHolder = value; }
    }

    Ссылок: 1
    public decimal Balance
    {
        get { return balance; } // Только для чтения
    }
```

```
// Конструктор для инициализации
Ссылок: 1
public BankAccount(string accountNumber, string accountHolder, decimal initialBalance)
{
    AccountNumber = accountNumber;
    AccountHolder = accountHolder;
    if (initialBalance > 0)
        balance = initialBalance;
    else
        throw new ArgumentException("Начальный баланс должен быть больше 0.");

}

// Метод для пополнения счёта
Ссылок: 1
public void Deposit(decimal amount)
{
    if (amount > 0)
    {
        balance += amount;
        Console.WriteLine($"Пополнение успешно! Текущий баланс: {balance}");
    }
    else
    {
        Console.WriteLine("Сумма пополнения должна быть положительной.");
    }
}
```

```
// Метод для снятия средств
Ссылок: 2
public void Withdraw(decimal amount)
{
    if (amount > 0 && amount <= balance)
    {
        balance -= amount;
        Console.WriteLine($"Снятие успешно! Текущий баланс: {balance}");
    }
    else
    {
        Console.WriteLine("Ошибка: недостаточно средств или некорректная сумма.");
    }
}

// Метод для отображения информации о счёте
Ссылок: 2
public void DisplayInfo()
{
    Console.WriteLine($"Номер счёта: {AccountNumber}");
    Console.WriteLine($"Владелец счёта: {AccountHolder}");
    Console.WriteLine($"Текущий баланс: {Balance}");
}
```

```
// Создаём новый банковский счёт
BankAccount account = new BankAccount("123456789", "Иван Иванов", 1000.00m);

// Отображаем информацию о счёте
account.DisplayInfo();
Console.WriteLine();

// Пополняем счёт
account.Deposit(500.00m);
Console.WriteLine();

// Пытаемся снять средства
account.Withdraw(300.00m);
Console.WriteLine();

// Пытаемся снять больше, чем есть на счёте
account.Withdraw(2000.00m);
Console.WriteLine();

// Итоговая информация
account.DisplayInfo();
```

Консоль отладки Microsoft Visual Studio

```
Номер счёта: 123456789
Владелец счёта: Иван Иванов
Текущий баланс: 1000,00

Пополнение успешно! Текущий баланс: 1500,00

Снятие успешно! Текущий баланс: 1200,00

Ошибка: недостаточно средств или некорректная сумма.

Номер счёта: 123456789
Владелец счёта: Иван Иванов
Текущий баланс: 1200,00
```

Список литературы:

1. Классы и объекты
2. Общие сведения о классах
3. Создание классов. Введение в C# ООП
4. Что такое класс | ООП C#

Материалы лекций:

https://github.com/ShViktor72/Education

Обратная связь:

colledge20education23@gmail.com