Введение в ООП

ООП (объектно-ориентированное программирование) — это парадигма программирования, при которой программа строится как набор взаимодействующих **объектов**.

Каждый объект объединяет в себе:

- **данные** (состояние, хранящееся в полях),
- **поведение** (методы, которые работают с этими данными).

Вместо того чтобы писать программу как последовательность процедур и работы с разрозненными данными, в ООП мы описываем **сущности реального мира или логические части системы** в виде классов, а затем создаём их конкретные реализации — объекты.

Основные принципы ООП

1. **Инкапсуляция** — объединение данных и методов, которые с ними работают, в одном объекте.
2. **Наследование** — возможность создавать новые классы на основе уже существующих.
3. **Полиморфизм** — возможность вызывать один и тот же метод у разных объектов, при этом выполняться он будет по-разному.
4. **Абстракция** — выделение только важных характеристик объекта и скрытие ненужных деталей.

Класс

- **Класс** — это **шаблон**, по которому создаются объекты. Он описывает, какие данные и какие действия у объекта будут.
- **Экземпляр (объект)** — это уже конкретная реализация класса, созданная в памяти с помощью ключевого слова `new`.

Пример:

```
class Car
{
    string brand;
    int year;
}
```

Тут `Car` — это класс.
А если мы пишем:

```
Car myCar = new Car();
```

— то `myCar` это **объект** (экземпляр класса `Car`).

Поля

Поле — это переменная, объявленная внутри класса. Она описывает данные (состояние), которые принадлежат каждому объекту этого класса.

Пример:

```
class Person
{
    string name; // поле
    int age;     // поле
}
```

Каждый объект `Person` будет иметь **свои собственные** значения `name` и `age`.

Сравнение:

- переменная `int x;` существует в методе,
- поле `int age;` существует внутри объекта и живёт столько же, сколько объект.

Конструктор

Конструктор — это специальный метод, который вызывается при создании объекта через `new`.
Обычно он используется, чтобы инициализировать поля (задать им стартовые значения).

```
class Person
{
    string name;
    int age;

    // конструктор
    public Person(string name, int age)
    {
        this.name = name; // this указывает, что это поле объекта
        this.age = age;   // справа — параметр метода
    }
}
```

`this`

Ключевое слово `this` — это ссылка на **текущий объект**.
Оно особенно полезно, когда имена параметров совпадают с именами полей.

Метод

Метод — это функция, которая объявлена внутри класса.
Методы описывают, что объект умеет делать.
Метод может использовать поля объекта (через `this` или напрямую).

```
class Person
{
    string name;
    int age;

    public void Greet()
    {
        Console.WriteLine($"Hi, I'm {name}, age {age}");
    }
}
```

Пример целиком

```
class Person
{
    // поля
    string name;
    int age;
    // конструктор
    Person(string name, int age)
    {
        this.name = name; // связываем параметры с полями объекта
        this.age = age;
    }
    // метод
    void Introduce()
    {
        Console.WriteLine($"Hello, I'm {name} and I'm {age} years old.");
    }
}

// Использование
var p1 = new Person("Anna", 20);
var p2 = new Person("Ivan", 30);

p1.Introduce(); // Hello, I'm Anna and I'm 20 years old.
p2.Introduce(); // Hello, I'm Ivan and I'm 30 years old.
```

Здесь видно:

- У каждого объекта свои данные (`Anna` , `Ivan`).
- Метод `Introduce` один и тот же для всех, но выводит разный результат, потому что использует разные значения полей.

2. Статические поля и методы

Что такое статические члены

- **Статическое поле или метод (`static`)** принадлежит **классу**, а не конкретному объекту.
  У класса есть только **одна копия** статического поля, которую используют все объекты этого класса.
- **Динамическое поле или метод** принадлежит конкретному объекту. У каждого экземпляра свои значения и своё поведение.

Пример: статическое и динамическое поле

```
class Counter
{
    static int TotalCount = 0; // общее поле для всего класса
    int InstanceId;            // поле экземпляра
    Counter()
    {
        TotalCount++;         // увеличиваем общий счётчик
        InstanceId = TotalCount; // у каждого объекта своё значение
    }
}
```

- `Counter.TotalCount` — общее для всех объектов.
- `InstanceId` — уникально для каждого.

```
var c1 = new Counter();
var c2 = new Counter();

Console.WriteLine(Counter.TotalCount); // 2
Console.WriteLine(c1.InstanceId);      // 1
Console.WriteLine(c2.InstanceId);      // 2
```

Статические и динамические методы

- **Статический метод** вызывается через имя класса: `ClassName.Method()`.
  Он **не имеет доступа** к полям объекта, потому что у него нет `this`.
- **Динамический метод** вызывается из экземпляра: `obj.Method()`.
  Он может использовать поля объекта напрямую.

Пример: статический и динамический метод, выполняющие одну задачу

```
class Messenger
{
    string _name;

    Messenger(string name)
    {
        _name = name;
    }

    // динамический метод: использует поле объекта напрямую
    void SayHello()
    {
        Console.WriteLine($"Hello from {_name}!");
    }

    // статический метод: ему нужно явно передать объект
    static void SayHelloStatic(Messenger m)
    {
        Console.WriteLine($"Hello from {m._name}!");
    }
}
```

Использование:

```
var m1 = new Messenger("Anna");
var m2 = new Messenger("Ivan");

m1.SayHello();                // Hello from Anna!
Messenger.SayHelloStatic(m2); // Hello from Ivan!
```

Разница:

- Динамический метод получает доступ к `_name` автоматически, через `this`.
- Статическому методу нужно передать объект вручную.

Таким образом, динамические методы — это **синтаксический сахар**, который упрощает работу с объектом.

Важно: под капотом

На самом деле в IL (байткоде .NET) **все методы статические**.
Для динамических методов компилятор автоматически добавляет скрытый параметр `this` — ссылку на объект, из которого вызывается метод.

Статический конструктор

Статический конструктор — это особый метод, который вызывается **один раз** перед первым использованием класса.

```
class Config
{
    static string AppName;

    // статический конструктор
    static Config()
    {
        AppName = "TodoApp";
        Console.WriteLine("Static constructor ran");
    }
}
```

Особенности:

- не принимает параметров;
- выполняется автоматически;
- запускается **до первого использования класса**.

Статический класс

- **Статический класс** (`static class`) — это класс, у которого **нельзя создать экземпляр** (`new` будет ошибкой).
- Он может содержать **только статические поля и методы**.
- Обычно используется для **глобальных вспомогательных функций** или сущностей, которые по своей природе существуют в единственном экземпляре (например `Console` , `Math`).

Пример

```
static class TimeUtils
{
    static DateTime StartOfDay = DateTime.Today; // статическое поле

    static string GetCurrentTime()
    {
        return DateTime.Now.ToString("HH:mm:ss");
    }

    static string GetDayDuration()
    {
        var now = DateTime.Now;
        var duration = now - StartOfDay;
        return $"С начала дня прошло {duration.Hours} ч {duration.Minutes} мин";
    }
}
```

Использование:

```
Console.WriteLine(TimeUtils.GetCurrentTime());   // "14:23:10"
Console.WriteLine(TimeUtils.GetDayDuration());   // "С начала дня прошло 2 ч 15 мин"
```

3. Целостность данных

Что такое целостность данных

Целостность данных — это гарантия, что внутреннее состояние объекта всегда остаётся корректным.
Класс сам контролирует, какие данные можно изменять и как именно.

Если не следить за целостностью данных, объект может перейти в "нелогичное" или "сломанное" состояние, и программа начнёт работать с ошибками.

Примеры проблем:

- У класса `BankAccount` есть публичное поле `decimal Balance`. Пользователь может присвоить `account.Balance = -1000;`, и в системе появится счёт с отрицательным балансом.
- У класса `Rectangle` есть два поля `Width` и `Height`. Если кто-то установит ширину < 0, то площадь прямоугольника станет бессмысленной.
- Несогласованные изменения: если объект хранит несколько взаимосвязанных полей, то при неправильном обновлении они могут "рассинхронизироваться".

Модификаторы доступа

Чтобы защитить данные, в C# используются **модификаторы доступа**:

- `public` — член класса доступен **откуда угодно**.
- `private` — член класса доступен **только внутри этого класса**.

Главное правило: **поля лучше всегда делать** `private`, а наружу давать доступ через **свойства или методы**.

Рекомендация по стилю: приватные поля часто называют с подчёркиванием: `_name`, `_balance`.
Это помогает:

- отличить их от параметров и свойств,
- не использовать лишний раз `this`.

Пример с модификаторами доступа

```
class BankAccount
{
    public decimal PublicBalance;   // открытое поле (плохая практика)
    private decimal _privateBalance; // приватное поле (правильный вариант)

    public BankAccount(decimal start)
    {
        PublicBalance = start;
        _privateBalance = start;
    }

    public void Deposit(decimal amount)
    {
        if (amount <= 0)
        {
            Console.WriteLine("Нельзя внести отрицательную сумму!");
            return;
        }
        _privateBalance += amount;
    }

    public void ShowBalances()
    {
        Console.WriteLine($"PublicBalance: {PublicBalance}, PrivateBalance: {_privateBalance}");
    }
}
```

Использование:

```
var acc = new BankAccount(100);

// public поле можно менять как угодно — нарушаем целостность
acc.PublicBalance = -5000; // ❌ неправильное состояние

// private поле напрямую поменять нельзя
// acc._privateBalance = -5000; // ошибка компиляции

// изменяем баланс только через метод
acc.Deposit(50); // ✅ корректное изменение
acc.ShowBalances();
```

Геттеры и сеттеры

Исторически для контроля над полями программисты писали методы вида `GetX()` / `SetX()`:

```
class Person
{
    private int _age;

    public int GetAge() => _age;

    public void SetAge(int value)
    {
        if (value < 0) throw new ArgumentException("Возраст не может быть отрицательным");
        _age = value;
    }
}
```

Такой подход позволяет:

- проверять корректность (валидацию),
- логировать изменения,
- ограничивать доступ к данным.

Свойства

В C# есть встроенный механизм для простой работы с геттерами и сеттерами — **свойства**.
Они позволяют писать код короче и чище, при этом оставляя возможность встроить логику.

Пример со свойством:

```
class Person
{
    private int _age;

    public int Age
    {
        get => _age;
        set
        {
            if (value < 0) throw new ArgumentException("Возраст не может быть отрицательным");
            _age = value;
        }
    }
}
```

Автосвойства

Если никакой логики в `get` и `set` нет, можно использовать короткую запись:

```
public string Name { get; set; }             // обычное свойство для чтения и записи
public string Id { get; }                    // только для чтения
public DateTime Created { get; private set; } // публичное чтение, приватная запись
```

Компилятор сам создаёт скрытое приватное поле, а вы работаете только со свойством.

`readonly`

Ключевое слово `readonly` используется для полей, которые должны быть заданы **один раз** — либо при объявлении, либо в конструкторе класса, и больше не могут изменяться.

Это помогает сохранить целостность данных: значение гарантированно не изменится случайно в коде.

Пример:

```
class Config
{
    public readonly string AppName = "TodoApp";    // задание при объявлении
    public readonly DateTime CreatedAt;            // задание в конструкторе

    public Config()
    {
        CreatedAt = DateTime.Now; // можно присвоить только здесь
    }
}
```

Использование:

```
var cfg = new Config();
Console.WriteLine(cfg.AppName);   // "TodoApp"
Console.WriteLine(cfg.CreatedAt); // дата создания объекта

// cfg.AppName = "OtherApp"; ❌ Ошибка: нельзя изменить readonly поле
```

4. Примеры

Пример рефакторинга: от процедурного к ООП

Исходный код (процедурный)

```
// Массивы с параллельными данными
string[] names = new string[100];
int[] ages = new int[100];
string[] groups = new string[100];
int studentCount = 0;

void AddStudent(string name, int age, string group)
{
    names[studentCount] = name;
    ages[studentCount] = age;
    groups[studentCount] = group;
    studentCount++;
}
void PrintAll()
{
    for (int i = 0; i < studentCount; i++)
    {
        Console.WriteLine($"{names[i]} (возраст {ages[i]}), группа {groups[i]}");
    }
}
```

```
// Использование
AddStudent("Иван", 20, "A1");
AddStudent("Анна", 19, "A1");
AddStudent("Пётр", 21, "B2");
PrintAll();
```

Проблемы такого подхода:

- три массива нужно поддерживать синхронно → легко ошибиться;
- добавить новый атрибут (например, email или оценки) — придётся переписывать весь код;
- логика и данные никак не объединены;
- код плохо читается и плохо расширяется.

Версия с ООП

```
class Student
{
    public string Name { get; }
    public int Age { get; }

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

    public void Print()
    {
        Console.WriteLine($"{Name} (возраст {Age})");
    }
}
```

Теперь объект сам хранит все свои данные и умеет себя печатать.
Использование стало проще:

```
var s1 = new Student("Иван", 20);
var s2 = new Student("Анна", 19);
var s3 = new Student("Пётр", 21);

s1.Print();
s2.Print();
s3.Print();
```

```
class Group
{
    private Student[] _students;
    private int _count;

    public string Name { get; }

    public Group(string name, int capacity = 30)
    {
        Name = name;
        _students = new Student[capacity];
        _count = 0;
    }

    public void AddStudent(Student student)
    {
        if (_count >= _students.Length)
            throw new InvalidOperationException("Группа переполнена");

        _students[_count++] = student;
    }

    public void PrintStudents()
    {
        Console.WriteLine($"Группа {Name}:");
        for (int i = 0; i < _count; i++)
            _students[i].Print();
    }
}
```

Использование:

```
var g1 = new Group("A1");
g1.AddStudent(new Student("Иван", 20));
g1.AddStudent(new Student("Анна", 19));

var g2 = new Group("B2");
g2.AddStudent(new Student("Пётр", 21));

g1.PrintStudents();
g2.PrintStudents();
```

Добавляем уровень «Курс»

```
class Course
{
    private Group[] _groups;
    private int _count;

    public string CourseName { get; }

    public Course(string courseName, int capacity = 10)
    {
        CourseName = courseName;
        _groups = new Group[capacity];
        _count = 0;
    }

    public void AddGroup(Group g)
    {
        _groups[_count++] = g;
    }

    public void PrintCourse()
    {
        Console.WriteLine($"Курс: {CourseName}");
        for (int i = 0; i < _count; i++)
            _groups[i].PrintStudents();
    }
}
```

Использование:

```
var course = new Course("Программирование на C#");
course.AddGroup(g1);
course.AddGroup(g2);

course.PrintCourse();
```

Что изменилось и почему стало лучше:

- **Инкапсуляция**: данные и методы, которые с ними работают, теперь в одном месте (`Student` , `Group` , `Course`).
- Добавить новое поле (`Email` , `Phone` , `AverageGrade`) можно в `Student` без переписывания всего кода.
- Код стал структурированным: у каждого уровня (студент, группа, курс) — своя зона ответственности.
- Легче читать и расширять.

5. Методы расширения

Методы расширения (extension methods) позволяют «добавить» новые методы к существующему классу, **не изменяя его исходный код**.
Это полезно, когда класс уже написан (например, из .NET библиотеки или чужого кода), но нам нужен дополнительный удобный метод.

Технически это обычный **статический метод** внутри **статического класса**.
Но благодаря тому, что первый параметр помечается ключевым словом `this` — этот метод можно вызывать так, как будто он встроен в объект.

Пример: метод расширения для `Student`

```
// Сам класс Student
class Student
{
    public string Name { get; }
    public int Age { get; }

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

```
// Статический класс с методом расширения
static class StudentExtensions
{
    // Добавляем метод IsAdult к Student
    public static bool IsAdult(this Student s)
    {
        return s.Age >= 18;
    }

    // Добавляем метод SayHello к Student
    public static void SayHello(this Student s)
    {
        Console.WriteLine($"Привет, меня зовут {s.Name}!");
    }
}
```

Использование:

```
var st = new Student("Анна", 20);

Console.WriteLine(st.IsAdult()); // True
st.SayHello(); // "Привет, меня зовут Анна!"
```

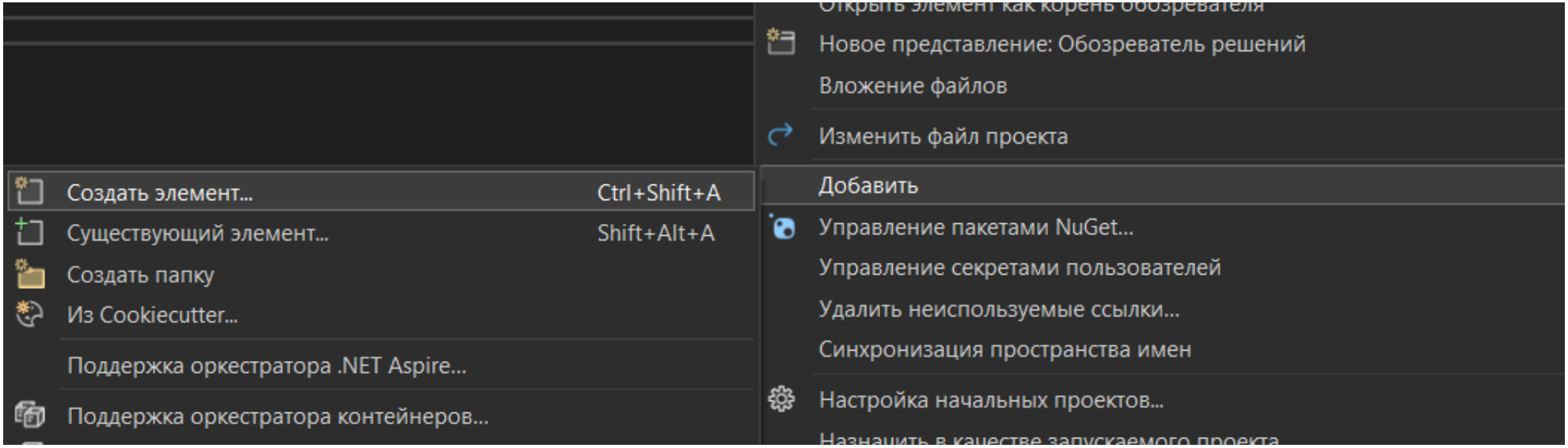
Практическое задание

Что нужно сделать:

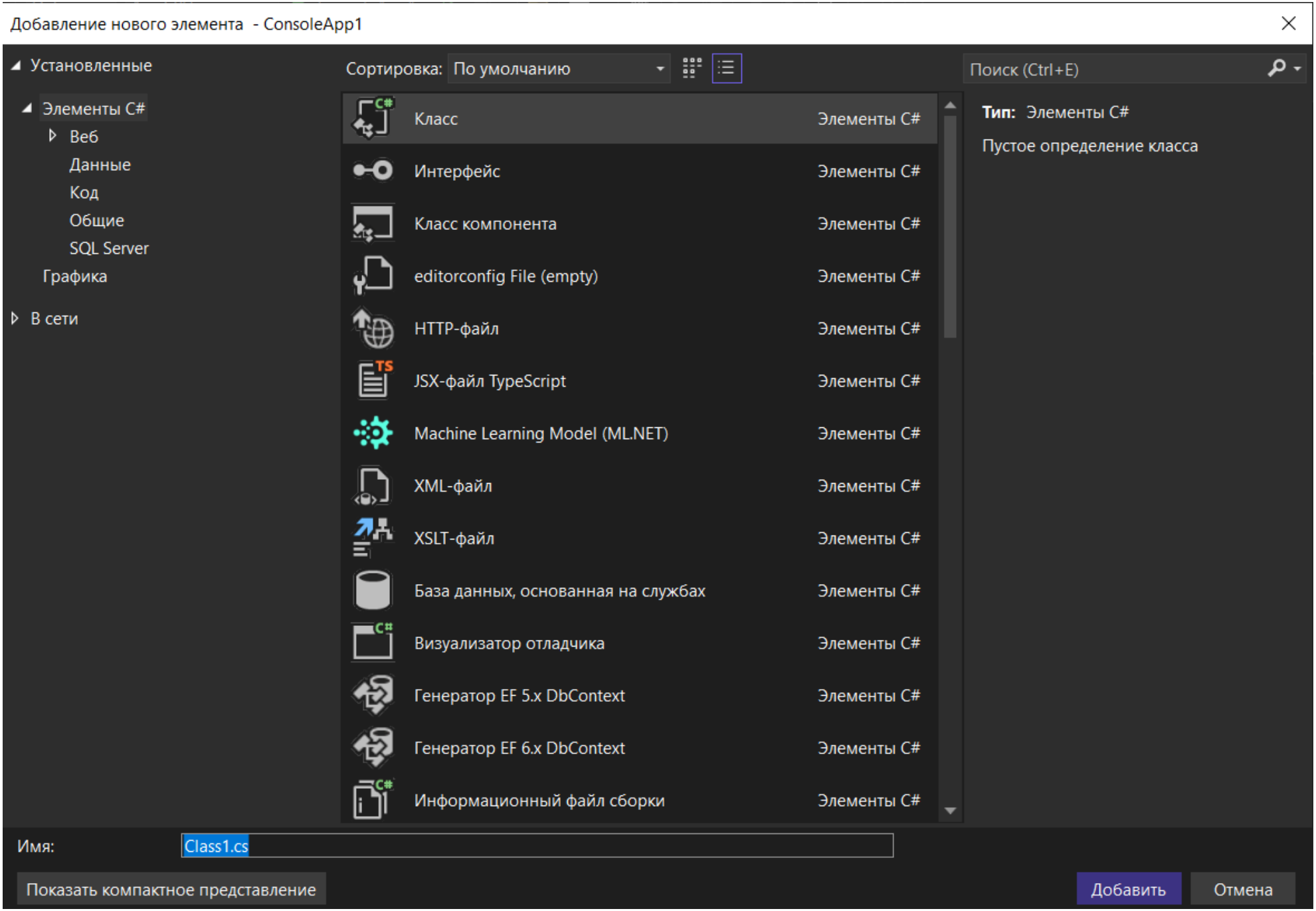
1. Продолжайте работу над проектом **TodoList**.
   На предыдущих шагах вы работали с массивами и функциями в процедурном стиле. Теперь нужно **переписать программу в объектно-ориентированном стиле**.
2. **Создайте класс `TodoItem`:**
   - Свойства:
     - `Text` — текст задачи;
     - `IsDone` — флаг выполнена/не выполнена;
     - `LastUpdate` — дата последнего изменения.
   - Конструктор принимает текст задачи, по умолчанию `IsDone = false`, `LastUpdate = DateTime.Now`.
   - Методы:
     - `MarkDone()` — отмечает задачу выполненной и обновляет дату.
     - `UpdateText(string newText)` — изменяет текст и обновляет дату.
     - `GetShortInfo()` — возвращает строку с краткой информацией (30 символов текста, статус, дата).
     - `GetFullInfo()` — возвращает полное описание.
3. **Создайте класс `TodoList`:**
   - Приватное поле: **массив** задач.
   - Методы:
   - `Add(TodoItem item)` — добавить задачу;
   - `Delete(int index)` — удалить задачу;
   - `View(bool showIndex, bool showDone, bool showDate)` — вывод задач в виде таблицы, в зависимости от флагов;
   - `GerItem(int index)` — получить задачу по индексу.
   - `private IncreaseArray(TodoItem[] items, TodoItem item)` — метод, который увеличивает размер массива при переполнении, используется только внутри класса.

4. **Создайте класс `Profile`:**
   - Свойства: `FirstName`, `LastName`, `BirthYear`.
   - Метод `GetInfo()` — возвращает строку `"Имя Фамилия, возраст N"`.
   - Должен использоваться для команды `profile`.
5. **Правила организации кода:**
   - Все новые классы (`TodoItem`, `TodoList`, `Profile`) создавайте **в отдельных файлах**.
   - Все поля внутри классов должны быть `private`, доступ извне должен происходить через свойства и методы.
6. **Измените логику программы так, чтобы:**
   - Команда `add` создаёт объект `TodoItem` и добавляет его в `TodoList`.
   - Команда `done <idx>` получает задачу через `GetItem()` и вызывает метод `MarkDone()`.
   - Команда `update <idx> "new text"` получает задачу через `GetItem()` и вызывает метод `UpdateText()`.
   - Команда `view` вызывает метод `View()` и выводит задачи в виде таблицы.
   - Команда `read <idx>` получает задачу через `GetItem()` и вызывает метод `GetFullInfo()`.
   - Команда `profile` выводит данные пользователя через `Profile.GetInfo()`.
7. Делайте коммиты после **каждого изменения**. Один большой коммит будет оцениваться в два раза ниже.
8. Обновите **README.md** — добавьте описание новых возможностей программы.
9. Сделайте push изменений в GitHub.

Для того, чтобы добавить новый файл с кодом для класса, нажмите на файл проекта правой кнопкой мыши, выберите `Добавить`, `Создать элемент`.



Выберите `Класс`, имя файлу дайте такое же как и у класса


Добавление нового элемента - ConsoleApp1
Установленные
Элементы C#
Веб
Данные
Код
Общие
SQL Server
Графика
В сети
Сортировка: По умолчанию
Поиск (Ctrl+E)
Класс Элементы C#
Интерфейс Элементы C#
Класс компонента Элементы C#
editorconfig File (empty) Элементы C#
HTTP-файл Элементы C#
JSX-файл TypeScript Элементы C#
Machine Learning Model (ML.NET) Элементы C#
XML-файл Элементы C#
XSLT-файл Элементы C#
База данных, основанная на службах Элементы C#
Визуализатор отладчика Элементы C#
Генератор EF 5.x DbContext Элементы C#
Генератор EF 6.x DbContext Элементы C#
Информационный файл сборки Элементы C#
Тип: Элементы C#
Пустое определение класса
Имя:
Class1.cs
Показать компактное представление
Добавить
Отмена