Методы, принципы проектирования и рефакторинг

1. Методы

Что такое метод?
Метод — именованный блок кода, выполняющий определённую задачу, это способ вынести повторяющийся или смысловой блок кода в отдельную единицу, чтобы:

- улучшить читаемость;
- убрать дублирование;
- разделить задачи на небольшие логические шаги;
- упростить тестирование отдельных операций.

Синтаксис

- `static` — специальное ключевое слово, смысл которого будет понятен на следующих лекциях.
- **Возвращаемый тип** — тип значения, которое метод возвращает (например, `int`, `string`, `bool`). Если метод ничего не возвращает, используется `void`. Возврат осуществляется через ключевое слово `return`.
- **Название метода** — должно отражать действие метода; рекомендуется начинать с глагола и использовать `PascalCase` (например, `CalculateSum`, `PrintReport`).
- **Сигнатура метода (параметры)** — перечисление входных параметров: для каждого указываются **тип** и **имя** (например, `int a`, `string name`). Параметры доступны внутри метода как локальные переменные.
- **Требование `return`** — если метод объявлен с возвращаемым типом (не `void`), он должен возвращать значение этого типа в соответствующих ветвях выполнения; иначе компилятор выдаст ошибку.
- **Вызов метода** — чтобы выполнить метод, пишут его имя и круглые скобки с аргументами (если есть), например: `Result(5, "text");`.

Пример

```
using System;

class Program
{
    static void Main()
    {
        // вызов методов
        PrintGreeting();
        int sum = Add(3, 5);
        Console.WriteLine(sum);
    }
    // void: ничего не возвращает
    static void PrintGreeting()
    {
        Console.WriteLine("Hello!");
    }
    // возвращает int, должен выполнить return
    static int Add(int a, int b)
    {
        return a + b;
    }
}
```

Методы с сокращённым синтаксисом

Вместо полного блока с `{ }` метод можно записать в одну строку с помощью стрелки `=>`. Такой метод сразу возвращает результат выражения.

Пример:

```
int Square(int x) => x * x;
bool IsEven(int n) => n % 2 == 0;
```

Когда использовать:

- метод выполняет простое действие в **одну строку**;
- метод возвращает результат вычисления без дополнительных шагов;
- для улучшения читаемости и компактности кода.

Когда не использовать:

- если внутри метода нужна более сложная логика (несколько строк кода, условия, циклы).

Параметры со значениями по умолчанию

В C# можно задать **значение по умолчанию** для параметра метода. Если при вызове метода аргумент не передан — используется это значение.

Синтаксис:

```
void PrintMessage(string text = "Hello", int count = 1)
{
    for (int i = 0; i < count; i++)
        Console.WriteLine(text);
}
```

Использование:

```
PrintMessage();                  // выведет "Hello" один раз
PrintMessage("Привет");          // выведет "Привет" один раз
PrintMessage("Hi", 3);           // выведет "Hi" три раза
```

Переменное число аргументов

`params` — ключевое слово в C#, позволяющее методу принимать **переменное число аргументов** одного типа. По сути это упрощённый способ передать в метод либо отдельные значения, либо уже готовый массив.

Ключевые правила

- Синтаксис: `type[]` с ключевым словом `params`, например `params int[] numbers`.
- Параметр `params` **должен быть последним** в списке параметров метода.
- В методе может быть **только один** параметр с `params`.
- `params` обязателен для одномерного массива (например, `params int[]`, `params string[]`).
- При вызове можно передать либо список значений, либо уже готовый массив.

Рассмотрим для примера метод, который сможет рассчитать сумму произвольного количества чисел:

```
static int Sum(params int[] values)
{
    int s = 0;
    foreach (var v in values) s += v;
    return s;
}

Console.WriteLine(Sum(1, 2, 3));        // 6
Console.WriteLine(Sum(10));             // 10
Console.WriteLine(Sum());               // 0
int[] arr = { 4, 5, 6 };
Console.WriteLine(Sum(arr));            // 15  (передали массив напрямую)
```

Возможные проблемы с ветвлением

При использовании ветвления внутри метода, у вас может возникнуть ошибка **«не все пути возвращают значение».** Если метод объявлен с возвращаемым типом (не `void`), компилятор требует, чтобы **каждый** возможный путь выполнения возвращал значение этого типа, либо бросал исключение. Если есть путь, где выполнение достигает конца метода без `return` появляется ошибка.

Частые причины

- `if` без `else` (некоторая ветка не возвращает значение); Чтобы исправить нужно добавить финальный `return`:

```
int GetSign(int x)
{
    if (x > 0) return 1;
    if (x < 0) return -1;
    return 0; // покрыли случай x == 0
}
```

- `switch` не покрывает все значения и нет `default`; Чтобы исправить нужно покрыть все варианты или добавить `default`:

```
string Name(Color c) => c switch
{
    Color.Red => "Red",
    Color.Green => "Green",
    Color.Blue => "Blue",
    _ => "Default color"
};
```

Перегрузка методов

Перегрузка методов (method overloading) означает создание несколько методов **с одним именем**, но с разными списками параметров. Компилятор выбирает подходящую версию по аргументам во время компиляции. Это называется **ad-hoc-полиморфизм**.

Правила и синтаксис

```
void Print(int x) { ... }
void Print(string s) { ... }
void Print(int a, int b) { ... }
```

- Сигнатура метода включает имя + типы и порядок параметров; **возвращаемый тип не входит** в сигнатуру — перегрузить только по возвращаемому типу нельзя.
- Разные варианты: разные типы параметров, разное число параметров, `params`-параметр, различают сигнатуры.

Примеры и поведение перегрузки

```
void Foo(int x) => Console.WriteLine("int");
void Foo(double x) => Console.WriteLine("double");

Foo(1);    // "int" — точное совпадение
Foo(1.0);  // "double"
Foo(1f);   // может выбрать double через преобразование (если нет float-версии)
```

Перегрузка выбирается по следующему правилу: сначала ищут точное совпадение, затем допустимые неявные преобразования, `params` и т.п.

Пример с `params`:

```
void Sum(int a, int b) => Console.WriteLine("pair");
void Sum(params int[] arr) => Console.WriteLine("many");

Sum(1,2);       // "pair" — более точный матч
Sum(1,2,3);     // "many"
```

Область видимости переменной

Общее правило: переменная доступна только **внутри фигурных скобок**, в которых она объявлена, и во **вложенных** скобках; **вне** этих скобок — недоступна.

Что важно знать

- Локальная переменная, объявленная в блоке `{ ... }`, видна внутри этого блока и во всех вложенных блоках.
- Переменная **не видна** за пределами блока, где она объявлена — попытка обратиться к ней вызовет ошибку компиляции.
- Можно заменить переменную: объявить переменную с тем же именем во вложенном блоке — внутри вложенного блока видна новая переменная, снаружи — старая.
- Параметры метода ведут себя как локальные переменные — их область видимости — тело метода.
- Поля класса (переменные вне методов) видны во всех методах.

Примеры

1. Блок и вложенный блок:

```
{
    int x = 5;
    {
        Console.WriteLine(x); // OK: видим x (вложенный блок)
    }
}
Console.WriteLine(x); // Ошибка: x не доступна вне внешнего блока
```

2. Замена переменной:

```
int a = 1;
{
    int a = 2; // компилятор C# не позволит объявить локальную переменную с таким же именем в той же области видимости,
               // но внутри вложенного блока это допустимо — она заменит внешнюю
    Console.WriteLine(a); // 2
}
Console.WriteLine(a); // 1 (внешняя переменная)
```

3. `for` — переменная цикла локальна:

```
for (int i = 0; i < 3; i++)
{
    Console.WriteLine(i); // i видна здесь
}
Console.WriteLine(i); // Ошибка: i недоступна вне цикла
```

4. Поля класса — видны в методах:

```
class C
{
    private int x = 10; // поле
    void M()
    {
        Console.WriteLine(x); // OK — поле доступно в методе
    }
}
```

6. Параметры метода, доступны только внутри него

```
class Program
{
    static void Main()
    {
        PrintSum(3, 5);

        // Ошибка: параметры метода недоступны вне метода
        Console.WriteLine(a); // нельзя — a не определена в этой области
    }

    static void PrintSum(int a, int b) // a и b — параметры, видимые внутри тела метода
    {
        int sum = a + b;
        Console.WriteLine($"Sum = {sum}"); // ОК: доступ к a и b

        if (sum > 5)
        {
            // параметры видны и во вложенных блоках
            Console.WriteLine($"a = {a}, b = {b} (внутри if)");
        }
    }
    // Параметры остаются локальными для метода:
    // другой метод не видит их
    static void Another()
    {
        Console.WriteLine(a); // ошибка: a не существует в этом методе
    }
}
```

2. Коротко о паттернах/принципы: DRY, KISS, SRP, YAGNI

DRY — *Don’t Repeat Yourself*

Идея: не дублировать логику. Если вы видите тот же кусок кода дважды — вынесите его в метод.
Почему: облегчает изменения (меняем в одном месте), уменьшает ошибок.
Пример применения: общий код валидации строки — вынести `bool ValidateName(string s)`.

KISS — *Keep It Simple, Stupid*

Идея: делайте код простым, избегайте ненужной сложности.
Почему: простой код легче понимать и поддерживать.
Применение: предпочитайте понятные конструкции, небольшие методы (5–20 строк).

SRP — *Single Responsibility Principle* (принцип единственной ответственности)

Идея: каждый метод должен решать ровно одну задачу.
Почему: упрощает тестирование и рефакторинг.
Пример: метод `ReadUser()` — читает данные; метод `CalculateAge()` — считает возраст; не совмещайте в одном методе чтение, валидацию и сохранение.

YAGNI — *You Aren’t Gonna Need It*

Идея: не добавляйте фичи «на всякий случай». Реализуйте только то, что нужно сейчас.
Почему: уменьшает сложность и технический долг.
Применение: не усложняйте интерфейсы методов и не добавляйте параметры «для будущих случаев».

3. Рефакторинг — что это и зачем

Рефакторинг — изменение внутренней структуры кода без изменения его внешнего поведения (функциональности). Цели:

- повысить читаемость,
- уменьшить дублирование,
- улучшить поддержку и тестируемость.

Когда рефакторить?

- перед добавлением новой функциональности (чтобы упростить её внедрение);
- после обнаружения дублирования;
- когда метод/файл становится слишком длинным.

Инструменты Visual Studio

- **Rename** — переименование символа с обновлением всех ссылок. `Ctrl + R, R`
- **Extract Method** — выделить блок кода в отдельный метод. `Ctrl + R, M`
- **Extract Variable** — вынести выражение в переменную. `Ctrl + R, V`

4. Практический разбор: рефакторинг «грязного» кода шаг за шагом

Рассмотрим как делать рефакторинг кода на примере программы, которая читает строку слов, разделённых пробелами, и считает:

- сколько слов длиннее 7 символов,
- сколько слов короче 3 символов,
- какое слово самое длинное и самое короткое.

```
class Program
{
    static void Main()
    {
        Console.WriteLine("Введите слова через пробел:");
        string s = Console.ReadLine();
        string[] a1 = s.Split(' ');
        int longC = 0;
        for (int i = 0; i < a.Length; i++)
        {
            if (a[i] != "")
            {
                string w = a[i].Trim(' ', ',', '.', '–', '!', '?', ';', ':', '(', ')', '[', ']', '{', '}');
                if (w.Length > 7) longC++;
            }
        }
        string[] a2 = s.Split(' ');
        int shortC = 0;
        for (int i = 0; i < a.Length; i++)
        {
            if (a[i] != "")
            {
                string w = a[i].Trim(' ', ',', '.', '–', '!', '?', ';', ':', '(', ')', '[', ']', '{', '}');
                if (w.Length < 3) shortC++;
```

```
            }
        }
        string[] a3 = s.Split(' ');
        string L = "";
        bool f = true;
        for (int i = 0; i < a.Length; i++)
        {
            if (a[i] != "")
            {
                string w = a[i].Trim(' ', ',', '.', '-', '!', '?', ';', ':', '(', ')', '[', ']', '{', '}');
                if (f) { L = w; f = false; }
                else if (w.Length > L.Length) L = w;
            }
        }
        string[] a4 = s.Split(' ');
        string S = "";
        f = true;
        for (int i = 0; i < a.Length; i++)
        {
            if (a[i] != "")
            {
                string w = a[i].Trim(' ', ',', '.', '-', '!', '?', ';', ':', '(', ')', '[', ']', '{', '}');
                if (f) { S = w; f = false; }
                else if (w.Length < S.Length) S = w;
            }
        }
        Console.WriteLine("Long words (>7): " + longC);
        Console.WriteLine("Short words (<3): " + shortC);
        Console.WriteLine("Longest: " + L);
        Console.WriteLine("Shortest: " + S);
    }
}
```

Проблемы исходного кода

- **Многократный разбор входной строки (`Split`)** — четыре/пять раз делаем одно и то же, лишняя работа.
- **Повторяющийся `Trim` / проверки пустой строки** — дублирование логики.
- **Магические числа**: `3`, `7`, а также литералы — непонятно без комментария.
- **Плохие имена** переменных (`s`, `a1`, `a2`, `L`, `S`, `f`) — ухудшают понимание.
- **Main слишком громоздкий** — чтение, парсинг, анализ и вывод смешаны в одном месте.
- **Нарушение DRY и SRP** — одинаковая логика повторяется в нескольких местах.

Шаги рефакторинга

Шаг 1 — вынести константы (убираем магические значения)

```
const int ShortLimit = 3;
const int LongLimit = 7;
const char Separator = ' ';
char[] ExtraChars = { ' ', ',', '.', '–', '!', '?', ';', ':', '(', ')', '[', ']', '{', '}' };
```

Шаг 2 — парсим строку один раз и очищаем токены

Вынесем парсинг в `ParseWords`, который делает `Split` с `RemoveEmptyEntries` и `Trim` — получаем единый массив `words`.

```
static string[] ParseWords(string text, char separator, char[] extraChars)
{
    string[] tokens = text.Split(separator);
    for (int i = 0; i < tokens.Length; i++)
        tokens[i] = tokens[i].Trim(extraChars);
    return tokens;
}
```

Использование в `Main`:

```
string input = s;
string[] words = ParseWords(input, Separator, ExtraChars);
```

Шаг 3 — вынести вспомогательные функции

Вынесем классификацию в отдельный метод `ClassifyLength`, он будет возвращать `LengthClass`. `LengthClass` — это `enum` с понятными названиями классов слов.

```
enum LengthClass { Short, Medium, Long }

static LengthClass ClassifyLength(string word, int shortLimit, int longLimit)
{
    if (word.Length < shortLimit) return LengthClass.Short;
    if (word.Length > longLimit) return LengthClass.Long;
    return LengthClass.Medium;
}
```

Шаг 4 — один проход по `words`, собираем всё

Вместо четырёх проходов — один `foreach`, где одновременно считаем все метрики и находим min/max.

```
int countShort = 0, countLong = 0;
string longest = words[0], shortest = words[0];

foreach (var word in words)
{
    var wordClass = ClassifyLength(word, ShortLimit, LongLimit);
    if (wordClass == LengthClass.Short) countShort++;
    if (wordClass == LengthClass.Long) countLong++;

    if (word.Length > longest.Length) longest = word;
    if (word.Length < shortest.Length) shortest = word;
}
```

Шаг 5 — вынести печать результата

```
static void PrintSummary(int total, int shortCount, int longCount, string longest, string shortest)
{
    Console.WriteLine("Total words: " + total);
    Console.WriteLine("Short words (<3): " + shortCount);
    Console.WriteLine("Long words (>7): " + longCount);
    Console.WriteLine("Longest: " + longest);
    Console.WriteLine("Shortest: " + shortest);
}
```

Финальный чистый код

```
enum LengthClass { Short, Medium, Long }

class Program
{
    static void Main()
    {
        const int ShortLimit = 3;
        const int LongLimit = 7;
        const char Separator = ' ';
        char[] ExtraChars = { ' ', ',', '.', '-', '!', '?', ';', ':', '(', ')', '[', ']', '{', '}' };

        Console.WriteLine("Введите слова через пробел:");
        string input = Console.ReadLine();
        string[] words = ParseWords(input, Separator, ExtraChars);

        if (words.Length == 0) return;

        int countShort = 0, countLong = 0;
        string longest = words[0], shortest = words[0];

        foreach (var word in words)
        {
            var classWord = ClassifyLength(word, ShortLimit, LongLimit);
            if (classWord == LengthClass.Short) countShort++;
            if (classWord == LengthClass.Long) countLong++;
            if (word.Length > longest.Length) longest = word;
            if (word.Length < shortest.Length) shortest = word;
        }

        PrintSummary(words.Length, countShort, countLong, longest, shortest);
```

```
    }

    static string[] ParseWords(string text, char separator, char[] extraChars)
    {
        string[] tokens = text.Split(separator);
        for (int i = 0; i < tokens.Length; i++)
            tokens[i] = tokens[i].Trim(extraChars);
        return tokens;
    }

    static LengthClass ClassifyLength(string word, int shortLimit, int longLimit)
    {
        if (word.Length < shortLimit) return LengthClass.Short;
        if (word.Length > longLimit) return LengthClass.Long;
        return LengthClass.Medium;
    }

    static void PrintSummary(int total, int shortCount, int longCount, string longest, string shortest)
    {
        Console.WriteLine("Total words: " + total);
        Console.WriteLine("Short words (<3): " + shortCount);
        Console.WriteLine("Long words (>7): " + longCount);
        Console.WriteLine("Longest: " + longest);
        Console.WriteLine("Shortest: " + shortest);
    }
}
```

Практическое задание

Что нужно сделать:

1. Продолжайте работу в проекте **TodoList**, созданном ранее.
2. Проведите **рефакторинг кода**:
   - Дайте всем переменным **понятные имена** в едином стиле.
   - Уберите **повторы** и **избыточные конструкции**.
   - Все «магические значения» замените на **константы с осмысленными названиями**.
   - Вынесите из `Main` отдельные **методы**:
     - для обработки каждой команды,
     - для расширения массивов,
     - при необходимости — для других логических частей.
3. Добавьте ещё два массива:
   - `statuses` — массив логических значений (`true/false`), который хранит, выполнено ли задание.
   - `dates` — массив дат, в котором хранится дата создания или последнего изменения задачи (`DateTime.Now`).
4. Сделайте так, чтобы все массивы (`todos` , `statuses` , `dates`) изменялись и расширялись синхронно.
5. Измените команды:
   - `add` — добавляет задачу, одновременно:
   - записывает в `done` значение `false` ,
   - записывает в `dates` текущую дату.
   - `view` — выводит задачи в формате:
   `<индекс> <текст задачи> <сделано/не сделано> <дата>`

6. Добавьте новые команды:
   - `done <idx>` — отмечает задачу выполненной:
     - в `done` записывается `true`,
     - в `dates` обновляется текущая дата.
   - `delete <idx>` — удаляет задачу по индексу:
     - все элементы массивов, идущие после неё, сдвигаются влево на одну позицию.
   - `update <idx> "new_text"` — обновляет текст задачи:
     - в `todos` записывается новый текст,
     - в `dates` обновляется текущая дата.
7. Каждую новую команду выделите в отдельный метод.
8. Делайте коммиты после **каждого** изменения. Если всё задание будет отправлено одним коммитом, оно будет оцениваться в два раза ниже.
9. После завершения реализации протестируйте программу:
   - добавьте несколько задач,
   - выполните часть из них,
   - обновите и удалите несколько,
   - проверьте корректность работы всех команд.
10. Внесите изменения в **README.md** — добавьте описание новых возможностей программы.
11. Сделайте push изменений.