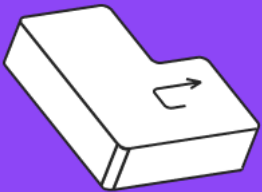


Как не нужно писать код

Основные рекомендации





Оглавление

[Приветствие](#)

[Почему важно понятно называть функции?](#)

[Разный код для одной задачи](#)

[Какую задачу решали разработчики?](#)

[Решение преподавателя](#)

[Что можно улучшить?](#)

[Общие правила понятного кода](#)

[Названия переменных](#)

[Нотации](#)

[Правила форматирования кода](#)

[Избегайте длинных строк](#)

[Используйте автоформатирование](#)

[Пример: упрощение кода](#)

[Используйте фишки языка](#)

[Bool](#)

[Красивый код: какой он?](#)

[Пример: использование неявной типизации](#)

[Продолжение. Красивый код: какой он?](#)

[Комментарии и документирование](#)

[Правила комментирования](#)

[Домашнее задание](#)

[00.01.36]

Приветствие

Друзья, всем привет! Мы продолжаем разбираться, как нужно писать код. Сегодня на лекции узнаем, как сделать его понятным не только для вас. Мы уже изучали язык C# и алгоритмы. Теперь научимся писать и оформлять код понятно. В качестве примеров возьмём код и названия методов, которые дают студенты (возможно, свой код вы здесь тоже увидите). А затем попробуем разобраться, стоит ли так писать. Давайте начинать.

[00.02.20]

Названия функций

Допустим, нескольким разработчикам дали задачу на написание кода. Первый дал методу имя Method, второй — ShowNumbers, третий — CreateArray, четвёртый — ShowNums, пятый — Ar, шестой — Numbers. Внимание, вопрос: какая задача решается, если методы названы таким образом?

[00.03.09]

Разный код для одной задачи

Зная названия, давайте посмотрим на реализацию этих методов. И, читая код, попробуем разобраться, что они делают.

Method принимает один аргумент и ничего не возвращает (или возвращает void). В нём определяется некоторое количество переменных, после чего производятся какие-то действия. В данном случае выводятся числа в консоль.

```
void Method(int maximum)
{
    int minimum;
    minimum = -maximum;
    while (minimum <= maximum)
    {
        Console.Write(minimum + " ");
        minimum++;
    }
}
```

CreateArray принимает в качестве аргумента число и возвращает массив. В теле метода создаётся массив определённого размера, затем цикл, который заполняет массив.

```
int[] CreateArray(int N)
{
    int[] arrayA = new int[N * 2 + 1];
    for (int i = -N; i <= N; i++)
    {
        arrayA[i + N] = i;
    }
    return arrayA;
}
```

Ar принимает одно число и возвращает другое. В теле метода производится перебор элементов от x , который в этом случае равен $-N$, до самого числа N , и затем что-то выводится в консоль. Дальше возвращается число x .

```
int Ar (int N) //задаем метод
{
    int x= -N; //первая цифра -N (задаем цикл)
    while (x <=N) //до тех пор пока x меньше или равен N
    {
        Console.WriteLine (x); //выводим в консоль "x"
        x++; //инкремент
    }
    return x;
}
```

Numbers — метод делает плюс-минус то же самое, только без возвращения значения. Просто в цикле в консоль выводятся формулы $(-n + i)$.

```
void Numbers(int n)
{
    int length = n + n;
    for (int i = 0; i < length + 1; i++)
    {
        Console.WriteLine(-n + i);
    }
}
```

ShowNums — возникает строка. Дальше цикл перебирает элементы от $-N$ до N . Причём здесь нет знака равенства, только знак меньше. В итоге возвращается та самая строка.

```
string ShowNums(int N)
{
    string NumsShow = "";
    for (int i = -N; i < N; i++)
    {
        NumsShow = NumsShow + i + " ";
    }
    return NumsShow;
}
```

[00.04.46]

Какую задачу решали разработчики?

Показать числа от $-N$ до N .

Какой из этих методов, на ваш взгляд, самый понятный с точки зрения описания? Что вам понравилось, а что нет?

[00.05.40]

Решение преподавателя

Не буду говорить, какой метод хорош, а какой плох, а предложу свой вариант решения:

```
int af = -5;
int uf = 5;
Console.WriteLine($"{af} .. {uf}");
```

Здесь есть одна переменная, равная какому-то значению, и другая — противоположная этому значению. В консоль выводится начальное значение, условное двоеточие и конечное значение. Ведь в задаче не сказано показать все числа — только числа «от». А что имелось в виду в этих «от и до» —

либо просто показать два числа и сообщить пользователь, что там будет какой-то дефис или двоеточие, либо показывать все числа.

Когда вы получаете задачу как разработчики из примера, лучше уточнять, что именно требуется. Возможно, показать весь диапазон чисел, возможно, только маленький кусочек.

Теперь давайте подумаем, что в текущем коде можно улучшить. Для этого разберём набор правил, которых нужно придерживаться при написании.

[00.06.52]

Что можно улучшить?

1. Имена переменных.
2. Имена методов.
3. Имена аргументов.

Может показаться, что третий пункт дублирует первый: имя аргументов и имя переменных — вроде бы одно и то же. Но это не так: для каждого пункта есть свой набор правил.

Есть и другие правила, но сейчас не о них, всё-таки мы изучаем не особенности языка C#, а учимся понимать общие принципы написания понятного кода.

Если вы выберете в качестве основного языка C#, почитайте статью от Microsoft об именовании ([Framework Design Guidelines](#)). Если Python или Java — изучите гайды.

[00.07.57]

Общие правила понятного кода

Код чаще читается, чем пишется. Он будет проходить код-ревью и тестирования. Корректное именование облегчит работу команде: тимлидам, ревьюерам, тестировщикам и техническим писателям, которые готовят документацию. Не нужно экономить на понятности и чистоте кода ради скорости.

Не используйте сокращения кроме общепринятых в продукте. Метод, который заполняет массив, можно назвать FillArray или Fa. С одной стороны, краткое название отражает суть, если мы знаем контекст, с другой — абсолютно непонятно для человека, незнакомого с кодом. Ему придётся тратить время на изучения тела метода, хотя его можно было просто понятно назвать.

Не используйте для нейминга зарезервированные слова. Они есть во всех языках программирования, причём в каждом свои. Зарезервированные слова C# или Swift отличаются от Python.

Пишите только на латинице и избегайте сложных слов, в том числе слов с удвоенными буквами и сложным чередованием согласных. Их тяжело читать и набирать.

Не используйте запрещённые слова даже в комментариях. Возможно, ваш проект выстрелит, станет open source и разлетится на цитаты. Или компания начнёт создавать по нему рекламные продукты. Подобная история произошла с Google — пользователи заметили брань в строках кода в официальной рекламе компании. Чтобы почитать подробнее, отсканируйте QR-код.



[00.11.38]

Названия переменных

Правильно компонуйте слова. Например, `HorizontalAlignment` — лучшее название идентификатора, чем `AlignmentHorizontal`, потому что читается легче.

Избегайте непонятных отсылок. Например, название `CanScrollHorizontally` лучше, чем `ScrollableX`, потому что во втором случае есть неочевидная ссылка на ось X. Она понятна человеку, который писал код, но непонятна тому, кто будет его изучать.

Не используйте символы, кроме букв и цифр (если это запрещено в языке). Например, в C# не приятно использовать символ нижнего подчёркивания. IDE для C# может подсказать, что в имени есть некорректный символ, а вот в Python всё неочевидно — можно понять, что есть ошибка, когда, казалось бы, написан работающий код.

Упомяну отдельно: Unity-разработчики в C# любят использовать для нейминга нижнее подчёркивание — `int_x`. Если мы говорим о классической разработке на .Net, так лучше не писать. Если об игровом движке Unity, для которого C# — не столько язык программирования, сколько скриптовый язык, правила могут разниться.

Избегайте идентификаторов, совпадающих с ключевыми словами популярных языков программирования. Почитать о них подробнее:

- для C# — [Ключевые слова C#](#),
- для Swift — [Lexical Structure](#).

Некоторые слова могут быть одинаковыми для нескольких языков, некоторые — нет.

Не сокращайте, если без этого можно обойтись. Если можно назвать идентификатор `GetWindow`, избегайте названий `GetWin`, `Gw` или `GetW`. Когда ваша программа будет собрана, эти имена компилятор всё равно переименует, как ему вздумается, — ему так удобнее. Сейчас я не буду показывать, во что компилятор превращает код, но на будущих лекциях мы немного об этом поговорим.

Не используйте акронимы. Вам или вашей команде они могут быть понятны, а вот заказчик запутается.

- `SendPM(int i, string t)` — название может запутать.
- `SendMessagePrivate(int id, string text)` — понятное название. Можно догадаться, что `id` — это идентификатор пользователя, `string text` — текст, который отправляют этому пользователю, а `SendMessagePrivate` — персональное сообщение.

Используйте универсальные имена платформы, не относящиеся к конкретному языку. Правило не распространённое, но подходит для платформы .Net и Java. В платформе есть свои наименования,

например типов данных. Для конкретного языка могут быть свои идентификаторы, которые принято использовать, когда вы пишете код. В платформе .Net есть идентификатор `int32`. В то же время в языке C# есть тип данных `int`. Если вы будете использовать, например, Visual Basic, там будет своё наименование, если Pascal — своё.

Используйте общие, не привязанные к контексту имена. `ConvertToByte(string value)` лучше, чем `ConvertToByte(string str)` — значение `value`, которое мы передаём в качестве аргумента, понятнее, чем ничего не отражающее `str`. Может показаться, что это сокращение типа данных, но, опять же, лучше писать `string value`, `string item` или `string element`, чем давать простое сокращение.

Именование методов аргументов, переменных и других системных единиц отличается. Может показаться, что имя (идентификатор) аргумента и переменная — одно и то же. Поле в C# тоже путают с переменной. Но у каждой системной единицы своё название и правила именования. Например, имена методов принято писать с заглавной буквы, а аргументов — со строчной. Всё это касается языка C#, подробнее об особенностях других языков — в гайдах.

[00.18.25]

Нотации

Мы разобрали перечень правил именования методов, полей, аргументов, переменных и более интересных системных единиц: например, структур записей, классов, пространства имён. Всё это определяется нотациями. В языке C# принято использовать две:

- **PascalCase** — составные слова пишутся слитно, первая буква каждого слова всегда заглавная: `BackColor`, `LastModified`, `DateTime`.
- **CamelCase** – всё так же, но первая буква первого слова — строчная: `borderColor`, `accessTime`, `templateName`.

Для именования методов используется что-то одно, для именования аргументов или переменных — другое. Когда вы начинаете разработку продукта (тем более в команде!), важно договориться, как вы будете давать названия и какие нотации станете использовать. Это нужно, что члены команды создавали свои модули так, чтобы при сборке получилось цельное приложение без разрозненности в коде.

[00.19.45]

Правила форматирования кода

Избегайте длинных строк

Разбивайте длинные конструкции на несколько строк. Например, можно по привычке написать длинную математическую формулу в одну строку, но она будет нечитаемой. Поэтому в некоторых командах принято договариваться о том, что длинные математические или логические выражения нужно разбивать на составные части и переносить каждую на новую строку.

```
int result = (1 + 2) - (3 + 4)
            / (15 * 16) - (17 + 19)
            / (25 * 26) - (27 + 29)
            / (35 * 36) - (37 + 39)
            ;
```

```
if (a > b
    && c > d
    && e > a
    || a == 1
    || c == f
    ^ !(g != h))
{
    Console.WriteLine("vse kruta");
}
```

На примере слева показано, как можно перенести простое арифметическое выражение. Вы можете сказать: «Здесь можно было вообще посчитать на калькуляторе и в качестве значения переменной `result` просто присвоить нужное число». Да, действительно можно. А теперь представьте, если в константных значениях мы бы использовали переменные или аргументы метода — всё бы выглядело уже не так просто.

В примере справа мы видим проверку достаточно сложного логического условия. Если бы оно было написано в одно строку, выглядело бы так себе. Причём, обратите внимание, именование аргументов — не то же самое, что вывод текста для конечного пользователя. В правом примере в консоль выводится фраза «vse kruta» — мне захотелось таким образом показать саму идею работы с UI, чтобы пользователь понимал, что же будет происходить. Но ваш код пользователи читать не будут, там всё-таки нужно придерживаться определённых правил.

В некоторых командных гайдлайнах указывают, что длина строки не может превышать 100–150 символов. Это нужно, чтобы на каждом мониторе код выглядел примерно одинаково. Например, если у вас широкоформатный 5K-монитор, на котором прекрасно видны даже 300 символов на строке, то у коллеги может быть простой Full HD, на котором не поместится то, что помещается на вашем.

В некоторых командах вместо отступа в 4 пробела используется табуляция. Дело в том, что, когда вы начинаете гонять свои исходники, может получиться, что замена этих 4-х пробелов одним табом существенно уменьшит вес исходников — иногда это критично.

[00.22.47]

Используйте автоформатирование

Мы используем Visual Studio Code, где отформатировать код можно сочетанием клавиш:

- В Windows — Shift + Alt + F.
- На Mac — Shift + Option + F.

Неотформатированный код выглядит плохо, поэтому используйте эти сочетания клавиш.

Сравните неотформатированный код и код с отступами, разнесённый на строки. Второй читается гораздо легче.


```

if (a>b
    && c>d
    && e> a
    || a ==1
|| c == f
    ^ !(g!= h))
{
    Console.WriteLine("vse kruta");
}

```

```

if (a > b
    && c > d
    && e > a
    || a == 1
    || c == f
    ^ !(g != h))
{
    Console.WriteLine("vse kruta");
}

```

[00.23.25]

Пример: упрощение кода

Давайте посмотрим, во что превратится метод, если не придерживаться правил:

```

bool Metodi(int chislo)
{
    bool resultati = false;
    if( chislo % 2==0 )
    { resultati = true;
      }
    else{resultati = false; }
    if (resultati == false)
    { return false;
      } else {
    return true;
    }
}

```

Этот метод в какой-то степени синтетический — я собрал несколько примеров кода воедино и дальше попытаюсь вам рассказать, почему так писать не нужно.

Во-первых видим, что не использовано форматирование — с ним читать код явно легче.

```

bool Metodi(int chislo)
{
    bool resultati = false;
    if (chislo % 2 == 0)
    {
        resultati = true;
    }
    else
    {
        resultati = false;
    }
    if (resultati == false)
    {
        return false;
    }
    else
    {
        return true;
    }
}

```

Но нужно ли здесь делать много переходов на новую строку? Скорее нет, потому что выполняется одна инструкция для каждого из `if` или соответствующей ветки `else`. То есть мы наблюдаем присваивание нужного значения переменной или возвращение результата работы функции.

```
bool Metodi(int chislo)
{
    bool resultati = false;

    if (chislo % 2 == 0) { resultati = true; }
    else { resultati = false; }

    if (resultati == false) { return false; }
    else { return true; }
}
```

Когда мы убрали эти строки, код становится компактнее, читать его проще. Разберём, что в нём происходит. Мы проверяем значение переменной `resultati`. Если `resultati = false`, возвращаем `false`, в противном случае — `true`.

Написанный код бывает полезно перечитывать на русском языке. Попробуем вместе: некоторый метод, который называется `Metodi`, в качестве аргумента принимает целое число — `chislo`. Тело метода — логическая переменная `resultati`, которой изначально присвоено значение лжи (`false`). Производится проверка: если остаток от деления входного числа на 2 равен нулю, значение переменной `resultati` становится истиной (`true`), в противном случае — ложью (`false`). Далее производится проверка: если результаты ложны, метод должен вернуть ложь, если нет — истину.

Согласитесь: когда читаем этот код, кажется, что что-то в нём лишнее. Давайте выясним, что. Например, что делает последняя проверка, последний `if`? Если результаты ложны, нужно вернуть ложь, иначе — истину. Тогда не проще ли просто вернуть значение переменной `resultati`?

```
bool Metodi(int chislo)
{
    bool resultati = false;

    if (chislo % 2 == 0) { resultati = true; }
    else { resultati = false; }

    return resultati;
}
```

Дальше подумаем, нужно ли вообще использовать переменную `resultati`? Теперь, если остаток от деления числа на 2 равен 0, мы просто будем возвращать истину. Если нет — ложь. Использование логической переменной абсолютно необоснованно.

```
bool Metodi(int chislo)
{

    if (chislo % 2 == 0) { return true; }
    else { return false; }

}
```

Код станет ещё проще, если мы уберём пустые строки.

```
bool Metodi(int chislo)
{
    if (chislo % 2 == 0) return true;
    else return false;
}
```

Дальше — результат деления числа на 2 либо истина (число разделилось), либо ложно (не разделилось). Что мы получаем? Что нам вообще не нужно делать проверку условия, достаточно написать в теле метода: «Верни результат проверки деления числа на 2».

```
bool Metodi(int chislo)
{
    return chislo % 2 == 0;
}
```

Осталось дать нормальное имя методу и аргументу.

```
bool IsEven(int value)
{
    return value % 2 == 0;
}
```

В итоге мы значительно сократили количество строк кода. Теперь понять и прочитать его гораздо проще.

До:

```
bool Metodi(int chislo)
{
    bool resultati = false;
    if (chislo % 2 == 0)
    {
        resultati = true;
    }
    else
    {
        resultati = false;
    }
    if (resultati == false)
    {
        return false;
    }
    else
    {
        return true;
    }
}
```

После:

```
bool IsEven(int value)
{
    return value % 2 == 0;
}
```

[00.27.17]

Используйте фишки языка

Изучайте фишки языка, на котором вы пишете — они тоже помогут упростить и сократить код. Например, если я использую фишки C#, код может стать таким:

```
* bool IsEven(int value) => value % 2 == 0;
```

Дальше — больше. Код можно переписать и так:

```
** var IsEven = (int v) => v % 2 == 0;
```

Для человека, который давно с C#, здесь всё понятно. Но для человека, который только входит в разработку, код может быть нечитаемым. Всё приходит с опытом. Когда вы выберете платформу, постепенно усвоите фишки языка (будь то C#, Swift, JavaScript или что-то ещё).

[00.28.15]

Bool

Не используйте проверки вида «логическое значение = false».

Плохо:

b == false

Хорошо:

!b

Лучше проверять на отрицание. В самом условии логического оператора это будет выглядеть лаконично и просто. Кому-то из вас может показаться, что текущий метод никому не нужен, пример синтетический, и никто в жизни не будет так писать. Это не так — я нашёл в репозитории JS [библиотеку](#), которая проверяет число на чётность. При этом её скачивают до 200 000 раз в неделю. Так что не исключено, что простой код, который вы напишете в дальнейшем, будет использоваться тысячами, а то и миллионами людей.

[00.29.07]

Красивый код: какой он?

Для методов используйте нотацию Pascal (вне зависимости от области видимости метода)¹.

GetStream()

Если в коде вы используете асинхронность, помечайте ключевым словом Async.

GetStreamAsync()

¹ В скобках, потому что пока мы разобрали не так много аспектов языка C#. В частности — что такое классы, записи, структуры или перечисления. Позже я буду показывать примеры того, как от того кода, который вы пишете, перейти к тому, который вы будете видеть в большинстве примеров, которые можно найти в первых ссылках Google.

Для именования переменных используйте нотацию Camel.

Используйте неявную типизацию (`var`), когда тип переменной понятен из правой части назначения или когда точный тип не важен. В языке C# есть неявная типизация. Её суть в том, что вместо какого-то типа данных вы можете написать ключевое слово `var`. Но C# — язык строго типизированный, поэтому не используйте неявную типизацию там, где она не нужна. Если знаете, что переменная будет целого типа, пишите `integer`, `long` или `pike`, но не `var`. Хотя в некоторых случаях в C# мы можем сконструировать что-то, для чего тип данных может быть неизвестен наперёд.

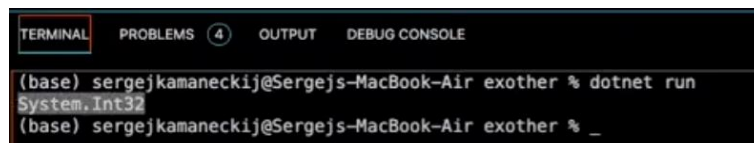
[00.31.03]

Пример: использование неявной типизации

Давайте для начала разберёмся, как можно узнать тип переменной. Возьмём переменную `a` со значением `12`. В консоли укажем `a` и через точку обратимся к методу `GetType`.

```
1 int a = 12;
2 Console.WriteLine(a.GetType());
```

Далее запустим код и обнаружим, что тип данных, который здесь описан, — это `System.Int32`.

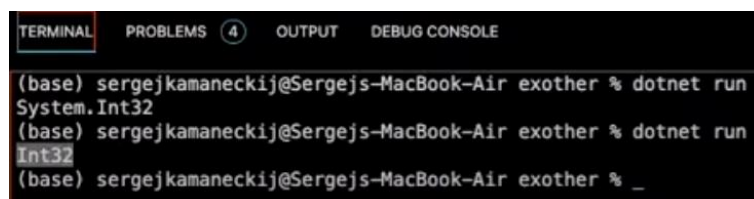


```
TERMINAL PROBLEMS (4) OUTPUT DEBUG CONSOLE
(base) sergejkamaneckij@Sergejs-MacBook-Air exother % dotnet run
System.Int32
(base) sergejkamaneckij@Sergejs-MacBook-Air exother % _
```

Чтобы не наблюдать это слово `System`, можно прописать более сложную конструкцию, например, `GetType.Name`.

```
1 int a = 12;
2 Console.WriteLine(a.GetType().Name);
```

Тогда в ответ мы получим только `Int32`. Но для нас это сейчас не особо важно.



```
TERMINAL PROBLEMS (4) OUTPUT DEBUG CONSOLE
(base) sergejkamaneckij@Sergejs-MacBook-Air exother % dotnet run
System.Int32
(base) sergejkamaneckij@Sergejs-MacBook-Air exother % dotnet run
Int32
(base) sergejkamaneckij@Sergejs-MacBook-Air exother % _
```

Смотрим дальше: мы технически можем вместо `int` написать ключевое слово `var`, от этого ничего не поменяется.

```
1 var a = 12;
2 Console.WriteLine(a.GetType());
```

Компилятор сразу поймёт, какие здесь данные. В нашем случае — `12`, целое число. Наведём на `var` курсор и увидим `System.Int32`, то есть тип данных определён корректно.

```
readonly struct System.Int32 );
```

Represents a 32-bit signed integer.

Теперь пример посложнее. Напишем массив с набором данных. Затем подключим системную библиотеку, чтобы произвести эту демонстрацию.

```
var a = 12;  
Console.WriteLine(a.GetType());  
  
var data = new int[]{1,2,3,4}.
```

Вам сейчас не нужно досконально понимать, что происходит в коде. Это всего лишь демонстрация того, что не нужно использовать `var` там, где можно обойтись без него.

Я хочу сделать выборку элементов, которые будут больше нуля. Далее хочу этот элемент превратить во что-то такое — e и $e+1$.

```
var a = 12;
Console.WriteLine(a.GetType());

var data = new int[] { 1, 2, 3, 4 }
    .Where(e => e > 0)
    .Select(e => new { e, e + 1 });
```

Добавим для них наименования, чтобы не было ошибки.

```
var a = 12;
Console.WriteLine(a.GetType());

var data = new int[] { 1, 2, 3, 4 }
    .Where(e => e > 0)
    .Select(e => new { q = e, w = e + 1 });
```

Код запускается. Делаем ставки на то, что будет, если я посмотрю тип данных переменной data.

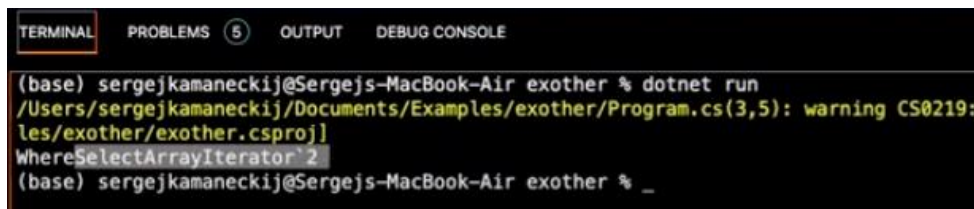
```

1 using System.Linq;
2
3 var a = 12;
4 //Console.WriteLine(a.GetType());
5
6 var data = new int[] { 1, 2, 3, 4 }
7     .Where(e => e > 0)
8     .Select(e => new { q = e, w = e + 1 });
9 Console.WriteLine(data.GetType().Name);

```

Попробуйте предположить, что мы в итоге получим.

Запустим код и посмотрим на тип данных, который мы получаем.

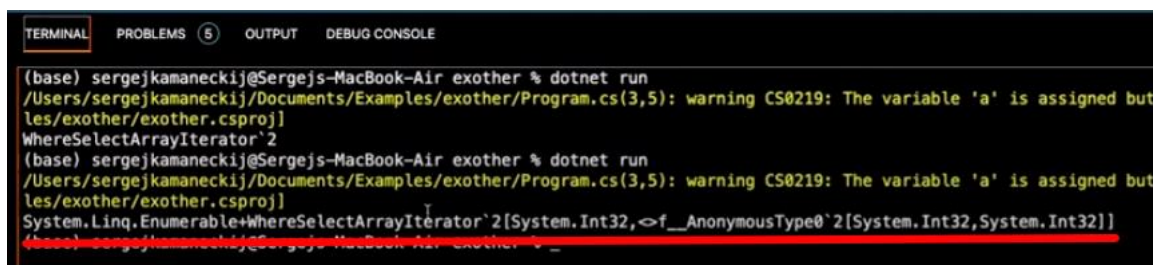


```

TERMINAL PROBLEMS (5) OUTPUT DEBUG CONSOLE
(base) sergejkamaneckij@Sergejs-MacBook-Air exother % dotnet run
/Users/sergejkamaneckij/Documents/Examples/exother/Program.cs(3,5): warning CS0219:
Variable 'a' is assigned but never used
WhereSelectArrayIterator`2
(base) sergejkamaneckij@Sergejs-MacBook-Air exother % _

```

Результат непонятный. Причём, если я уберу Name из 9-й строки, в консоли увидим следующее:



```

TERMINAL PROBLEMS (5) OUTPUT DEBUG CONSOLE
(base) sergejkamaneckij@Sergejs-MacBook-Air exother % dotnet run
/Users/sergejkamaneckij/Documents/Examples/exother/Program.cs(3,5): warning CS0219: The variable 'a' is assigned but
never used
WhereSelectArrayIterator`2
(base) sergejkamaneckij@Sergejs-MacBook-Air exother % dotnet run
/Users/sergejkamaneckij/Documents/Examples/exother/Program.cs(3,5): warning CS0219: The variable 'a' is assigned but
never used
System.Linq.Enumerable+WhereSelectArrayIterator`2[System.Int32, <f__AnonymousType0`2[System.Int32,System.Int32]]
(base) sergejkamaneckij@Sergejs-MacBook-Air exother % _

```

То есть, чтобы произнести такой тип, нам потребуется много времени. Так что в некоторых случаях var просто необходим, а когда-то без него точно можно обойтись. Просто помните об этом.

[00.35.18]

Продолжение. Красивый код: какой он?

Объявляйте переменные перед тем, как планируете их использовать. Пример плохой практики: в начале написать `int = a`, затем 200 строк кода и после них указать, что `a = 123`.

Так делать не нужно: пока вы (или кто-то другой) будете читать 200 строк кода, скорее всего, забудете, что эта переменная у вас объявлена. И, как следствие, может сложиться впечатление, что есть ошибка, и код не должен работать. Или захочется повторно прописать `int = a`, что вызовет ошибку.

Не нужно выдумывать сложные имена счётчиков, потому что это даже не второстепенная конструкция вашего кода. Это некоторая переменная, которая меняет свое значение «от» и «до», поэтому выдумать для неё сложные названия не стоит. *i*, *j*, *k*, *l*, *m*, *n* — это поймут все, когда у вас индекс, а если он ещё используется в теле цикла и будет назван непонятно как, такой код будет сложно читаться.

Инициализируйте переменные при объявлении, если такая возможность есть. Просто `var b` или `int b` — плохо. Компилятор не знает, сколько памяти выделять на хранение переменной.

Не используйте цепочки вызовов, где один метод использует результат работы другого — получается слишком много вызовов. Когда вы используете какой-то метод, который возвращает какие-то данные, и эти данные больше нигде не должны использоваться, мы можем одному методу передавать результат работы другого. Например:

```
PrintMatrix(  
    FillMatrix(  
        CreateMatrix(5, 5)),  
    10);
```

`CreateMatrix` передаётся в `FillMatrix`, который сразу же печатается методом `PrintMatrix`.

Если у нас 3 вызова — это нормально, но не стоит увлекаться и писать такие конструкции:

```
SaveToFile(  
    PrintArraySumElements(  
        SumElements(  
            GetArrayElements(  
                PrintMatrix(  
                    MultiplicationMatrixByNumber(  
                        PrintMatrix(  
                            FillMatrix(  
                                CreateMatrix(5, 5)),  
                                10))))));
```

Такой код не читаем. С языком `C#` легко обходить такие каскады вызовов, но для этого надо знать чуть больше, и чуть больше программировать.

Системные единицы должны находиться рядом. Не разбрасывайте по коду данные одного типа, скомпонуйте их в одном месте: где-то поля, где-то методы, где-то свойства и так далее. Системные единицы должны находиться рядом.

Также есть определённые правила для описания исключений, пространств, имён, классов, записей и так далее. В каждом языке есть много тонкостей, синтаксических конструкций и единиц, о которых нужно знать. Если вы будете переключаться между языками, в голове нужно будет держать, что сейчас вы пишете на `JS` с его набором правил, а через время переключитесь на `Java` с другим. Если через время ваша компания переедет на `.NET`, нужно будет учить новые гайдстайлы и придерживаться новых политик.

[00.40.17]

Комментарии и документирование

Я не сторонник таких комментариев, где описывается каждая строчка кода. Точно не нужно писать комментарии вроде таких: «объявляем переменную счётчика» или «выводим значение на экран». Но хорошо бы пометить, что концептуально делает ваш метод.

В определённых IDE, например, Visual Studio (не Visual Studio Code, которую мы сами используем) можно оставлять комментарии, которые попадут в список задач. Это полезно, если мы описываем метод, и у нас не хватает знаний или времени, чтобы закончить его сиюминутно. В этом случае пишем комментарий TODO, и он попадает в список задач, к нему можно будет вернуться позже и доделать. Такая возможность есть не во всех средах разработки.

[00.41.28]

Правила комментирования

Внутри блока с кодом отделяйте текст комментария одним пробелом. Например: `«// Текст комментария.»`

Неиспользуемый код не комментируйте, а удаляйте. Если вы закомментировали код, значит он не нужен и лучше его удалить, а не оставлять в исходниках, например, на GitHub-репозитории. Кстати, это намёк на случаи, когда вы описываете решение 20 задач в одном файле. Как всё реализовано в большинстве случаев? Есть метод, он отработал, вы его закомментировали и начинаете писать следующий. У вас 10 строчек кода для решения одной задачи, ещё 10 для другой задачи. Когда у вас список из 100 задач, и на каждую вы отводите 20 строчек кода, получается 2000 строк, и это выглядит так себе. В таком коде невозможно ориентироваться. Когда вы сами захотите найти конкретный метод, это будет непросто.

Есть негласное правило — старайтесь писать код так, чтобы один метод умещался на одну единицу экрана. Для этого и применяются стандарты: максимальная «ширина» строки — 130 символов, «высота» кода — 30 строк. Поэтому считается, что в файлах, в которых больше 120 строк кода на языке C#, нужно делать декомпозицию. Если метод не помещается на один экран, нужно подумать, как его разбить. Также с большими системными единицами, например классами. Старайтесь не делать всё в одном файле, разбивайте на меньшие сущности.

Если код комментируется временно, он должен быть с пометкой `«// TODO: причина»`. Как я уже говорил, задачи, которые должны будут выполнены позже, помечают тегом TODO.

В языке C# (в частности, в Visual Studio) есть возможность документирования кода. Это полезно для технических писателей — на основе сформированного файла они могут составлять документацию по проекту.

Используйте словари для подсветки ошибок на русском и английском. В Visual Studio (но, возможно, и в других средах разработки) можно использовать словари. Возможно, в коде, который написан, откомментирован и задокументирован, будет куча ошибок и опечаток. Поэтому к среде разработки полезно подключить словарь, который будет проверять орфографию и пунктуацию, подсвечивать возможные ошибки.

[00.45.00]

Домашнее задание

В качестве домашнего задания я предлагаю открыть свои Git-репозитории и посмотреть, насколько там всё хорошо или плохо.

Ещё интереснее будет, если в общем чате вы найдёте коллегу, который оценит ваш код, а вы — его. То есть вы сделаете небольшой код-ревью с описанием того, что и как нужно изменить.

Возможно, читать код будет сложно. Но мы учимся, поэтому не стоит расстраиваться. До сегодняшней лекции вам никто не рассказывал, что просто рабочего кода недостаточно — он ещё должен быть

понятен другим разработчикам. Теперь вы это знаете, поэтому старайтесь придерживаться правил, о которых мы сегодня узнали. К примеру, не пытайтесь всё описывать в одном методе. Если у вас метод, который одновременно и заполняет, и создаёт, и печатает массив, и сразу ищет там число больше нуля, это уже 4 задачи, 4 сущности, 4 метода, которые можно выделить по отдельности.

Спасибо, что уделите время для просмотра этой лекции. На этом я с вами прощаюсь. До встречи на семинаре!