Лекция

Тема: Стили в программировании

Работая с компьютерами и, в особенности, занимаясь программированием, вы неизбежно столкнетесь с необходимостью давать названия чему-либо. Чтобы именование было успешным, самое главное – определиться со стилем написания составных слов. Это важно для поддержания последовательности в пределах одного проекта или рабочего пространства. Если вы занимаетесь созданием программного обеспечения, то в спецификации своего языка вы можете встретить указание на определенный стиль. Некоторые языки очень полагаются на знание пользователя разницы между стилями и правильное их использование.

# Разберем самые популярные нотации

**Нотация** - соглашения об именовании переменных, констант и других идентификаторов в программном коде. Рассмотрим сначала с позиции языков программирования JavaScript, Python, а затем C#.

## camelCase

camelCase должен начинаться со строчной буквы, а первая буква каждого последующего слова должна быть заглавной. Все слова при этом пишутся слитно между собой. Пример camelCase для имени переменной camel case var – camelCaseVar.

**ПРИМЕР**

calculateElephantWeight

## snake_case

Чтобы писать в стиле snake_case, нужно просто заменить пробелы знаками подчеркивания. Все слова при этом пишутся строчными буквами. Можно использовать snake_case, смешивая его с camelCase и PascalCase, но, как по мне, при этом теряется сам смысл этого стиля. Пример snake_case для имени переменной snake case var – snake_case_var.

**ПРИМЕР**

calculate_elephant_weight

## kebab-case

kebab-case похож на snake_case, только в нем пробелы заменяются на дефисы. Слова также пишутся строчными буквами. Опять же, его можно смешивать с camelCase и PascalCase, но в этом нет смысла. Пример kebab-case для переменной kebab case var – kebab-case-var.

**ПРИМЕР**

calculate-elephant-weight

## PascalCase

В PascalCase каждое слово начинается с заглавной буквы (в отличие от camelCase, где первое слово начинается со строчной). Пример PascalCase для переменной pascal case var – PascalCaseVar.

Примечание: этот стиль часто путают с camelCase, но это, тем не менее, отдельный стиль.

**ПРИМЕР**

CalculateElephantWeight

## UPPER_CASE_SNAKE_CASE

В UPPER_CASE_SNAKE_CASE все слова пишутся заглавными буквами, а пробелы заменяются знаками подчеркивания.

Пример UPPER_CASE_SNAKE_CASE для переменной upper case snake case var – UPPER_CASE_SNAKE_CASE_VAR.

## Плоская нотация (flat case, flatcase)

Чтобы получить наименование в этом стиле, нужно просто записать слова рядом без пробелов, все буквы каждого слова должны быть строчными.

**ПРИМЕР**

calculateelephantweight

Переменные, классы и другие элементы программ обычно так не называют — их будет сложно разделить на слова при чтении, особенно если слов больше двух, как в примере. Зато плоская нотация встречается в именах пакетов. В Java, например, можно создать пакет com.example.flatcase.mypackage.

Но чаще всего такого рода длинные надписи мы видим в соцсетях — #этожеобычнаяпрактикадлятегов :)

## Как выбрать стиль написания составных слов?

Теперь, когда вы ознакомились с различными стилями, давайте рассмотрим примеры их использования для выбора названий файлов и для программирования на разных языках.

## Какого соглашения придерживаться, выбирая имена для файлов?

### Совет: всегда snake_case

При выборе названий для файлов важно задавать вопрос: «Каков наименьший общий знаменатель?». Если у вас нет собственного мнения на этот счет, поделюсь своим опытом. Лучший результат у меня всегда получался при использовании snake_case. В этом случае название файла сохраняет читабельность и при этом вряд ли приведет к каким-либо проблемам в файловой системе.

Если вы пользуетесь Mac или работаете с пользователями Mac, будет хорошей идеей всегда использовать только строчные буквы. Файловая система Mac – HFS+, а поскольку она нечувствительна к регистру, то файлы «MyFile» и «myfile» не будут различаться.

Мой главный аргумент в пользу этого подхода связан с особенно коварным «багом», который я видел при запуске CI/CD (непрерывной интеграции/непрерывной доставки) кластера. Во время сборки проекта на React в работе CI возник сбой с сообщением «файл не найден: mycomponent.js». Разработчик божился, что этот файл был в исходниках проекта.

Я обнаружил, что они импортировали «mycomponenet.js», но файл назывался «MyComponent.js». Это ведь был проект на React, где для наименований компонентов файлов используется именно PascalCase. Поскольку HFS+ не различает регистры, файл «MyComponent.js» был успешно принят за «mycomponent.js», когда разработчик писал код (на Mac). Но когда выполнялась сборка на сервере CI (а он был на основе Unix), возник сбой, потому что система искала точное соответствие названия.

## Соглашения JavaScript

- camelCase для переменных и методов.
- PascalCase для типов и классов.
- UPPER_CASE_SNAKE_CASE для констант.

## Соглашения Python

- snake_case для названий методов и переменных экземпляра (PEP8).
- UPPER_CASE_SNAKE_CASE – для констант.

### Другие соглашения

- kebab-case в Lisp.
- kebab-case в HTTP URL (most-common-programming-case-types/).
- snake_case в ключах свойств JSON.

## Таблица для быстрого сравнения

| Стиль написания | Пример |
|---|---|
| Исходное написание имени переменной | some awesome var |
| Camel Case | someAwesomeVar |
| Snake Case | some_awesome_var |
| Kebab Case | some-awesome-var |
| Pascal Case | SomeAwesomeVar |
| Upper Case Snake Case | SOME_AWESOME_VAR |

Теперь, зная самые распространенные стили написания, вам будет легче определиться, какой использовать при написании своего кода!

Разберем конкретно язык C#.

## Правила и соглашения об именовании идентификаторов C#

**Идентификатор** — это имя, присваиваемое типу (классу, интерфейсу, структуре, записи, делегату или перечислению), элементу, переменной или пространству имен.

Правила именования

Допустимые идентификаторы должны соответствовать следующим правилам.

- Идентификаторы должны начинаться с буквы или знака подчеркивания ( _ ).
- Идентификаторы могут содержать буквенные символы Юникода, десятичные числа, символы соединения Юникода, несамостоятельные знаки Юникода или символы форматирования Юникода. Дополнительные сведения о категориях Юникода см. в разделе База данных категорий Юникода. Вы можете объявить идентификаторы, соответствующие ключевым словам C#, с помощью префикса идентификатора `@`. `@` не является частью имени идентификатора. Например, `@if` объявляет идентификатор с именем `if`. Эти буквальные идентификаторы предназначены главным образом для взаимодействия с идентификаторами, объявленными в других языках.

Соглашения об именах

По соглашению программы C# используют `PascalCase` для имен типов, пространства имен и всех открытых членов. Кроме того, часто используются следующие соглашения.

- Имена интерфейсов начинаются с заглавной буквы `I`.
- Типы атрибутов заканчиваются словом `Attribute`.
- Типы перечисления используют единственное число для объектов, не являющихся флагами, и множественное число для флагов.

- Идентификаторы не должны содержать два последовательных символа подчеркивания ( _ ). Эти имена зарезервированы для создаваемых компилятором идентификаторов.

# Соглашения о написании кода на C#

Соглашения о написании кода предназначены для реализации следующих целей.

☞ Создание согласованного вида кода, позволяющего читателям сосредоточиться на содержимом, а не на структуре.

☞ Предоставление читателям возможности делать предположения, основанные на опыте, и поэтому быстрее понимать код.

☞ Упрощение процессов копирования, изменения и обслуживания кода.

☞ Предоставление лучших методик C#.

**Важно!**

Майкрософт использует приведенные в этой статье рекомендации для разработки примеров и документации. Они были приняты в **среде выполнения .NET, правила стиля написания кода C#** . Вы можете использовать их или адаптировать их к вашим потребностям. Основными целями являются согласованность и удобочитаемость в проекте, команде, организации или исходном коде компании.

## Соглашения об именах

Существует несколько соглашений об именовании, которые следует учитывать при написании кода C#.

В следующих примерах любое из указаний, относящихся к элементам, помеченным `public` , также применимо при работе с `protected` элементами и `protected internal` элементами, которые должны быть видны внешним вызывающим элементам.

## Регистр Pascal

Используйте регистр pascal ("PascalCasing") при именовании `class`, `record`или `struct`.

C#
```
public class DataService
{
}
```
C#
```
public record PhysicalAddress(
    string Street,
    string City,
    string StateOrProvince,
    string ZipCode);
```
C#
```
public struct ValueCoordinate
{
}
```
При именовании `interface`используйте регистр pascal в дополнение к префиксам имени с именем `I`. Это явно указывает потребителям, что это `interface`.

C#
```
public interface IWorkerQueue
{
}
```

При именовании `public` элементов типов, таких как поля, свойства, события, методы и локальные функции, используйте регистр pascal.

```C#
public class ExampleEvents
{
    // A public field, these should be used sparingly
    public bool IsValid;

    // An init-only property
    public IWorkerQueue WorkerQueue { get; init; }

    // An event
    public event Action EventProcessing;

    // Method
    public void StartEventProcessing()
    {
        // Local function
        static int CountQueueItems() => WorkerQueue.Count;
        // ...
    }
}
```

При записи позиционных записей используйте регистр pascal для параметров, так как это открытые свойства записи.

```C#
public record PhysicalAddress(
    string Street,
    string City,
    string StateOrProvince,
    string ZipCode);
```

## Верблюжья нотация

Используйте регистр верблюда ("camelCasing") при именовании `private` или `internal` полях, а также префикс их с _помощью .

```C#
public class DataService
{
    private IWorkerQueue _workerQueue;
}
```

**Совет**

При редактировании кода C#, следующего за этими соглашениями об именовании в интегрированной среде разработки, поддерживающей завершение инструкции, при вводе _ будут отображаться все члены области объекта.

При работе с полями `static`, которые являются `private` или `internal`, используйте `s_` префикс и для статического использования `t_`потока.

```C#
public class DataService
{
    private static IWorkerQueue s_workerQueue;

    [ThreadStatic]
    private static TimeSpan t_timeSpan;
}
```

При написании параметров метода используйте регистр верблюда.

```
C#
public T SomeMethod<T>(int someNumber, bool isValid)
{
}
```

Дополнительные сведения о соглашениях об именовании C# см. в разделе "Стиль написания кода C#".

## Дополнительные соглашения об именовании

- Примеры, не включающие директивы using, используйте квалификацию пространства имен. Если известно, что пространство имен импортируется в проект по умолчанию, вам не нужно указывать полные имена из этого пространства имен. Полные имена, если они слишком длинные для одной строки, можно разбить после точки (.), как показано в следующем примере.

```
C#
var currentPerformanceCounterCategory = new System.Diagnostics.
    PerformanceCounterCategory();
```

- Вам не нужно изменять имена объектов, созданных с помощью инструментов разработки Visual Studio, чтобы привести их в соответствие с другими рекомендациями.

## Соглашения о макете

Чтобы выделить структуру кода и облегчить чтение кода, в хорошем макете используется форматирование. Примеры и образцы корпорации Майкрософт соответствуют следующим соглашениям.

- Использование параметров редактора кода по умолчанию (логичные отступы, отступы по четыре символа, использование пробелов для табуляции). Дополнительные сведения см. в разделе "Параметры", "Текстовый редактор", C#, "Форматирование".
- Запись только одного оператора в строке.
- Запись только одного объявления в строке.
- Если отступ для дополнительных строк не ставится автоматически, необходимо сделать для них отступ на одну позицию табуляции (четыре пробела).
- Добавление по крайней мере одной пустой строки между определениями методов и свойств.
- Использование скобок для ясности предложений в выражениях, как показано в следующем коде.

```
C#
if ((val1 > val2) && (val1 > val3))
{
    // Take appropriate action.
}
```

## Соглашения о комментариях

- Комментарий размещается на отдельной строке, а не в конце строки кода.

- Текст комментария начинается с заглавной буквы.
- Текст комментария завершается точкой.
- Между разделителем комментария (// ) и текстом комментария вставляется один пробел, как показано в следующем примере.

C#
```
// The following declaration creates a query. It does not run
// the query.
```

- Не создавайте форматированные блоки из звездочек вокруг комментариев.
- Убедитесь, что все открытые члены имеют необходимые XML-комментарии, предоставляющие соответствующие описания их поведения.

Рекомендации по использованию языка

В следующих подразделах описаны методики, которыми руководствуется команда C# для подготовки примеров и образцов кода.

## Строковый тип данных

- Для сцепления коротких строк рекомендуется использовать интерполяцию строк, как показано в следующем коде.

C#
```
string displayName = $"{nameList[n].LastName}, {nameList[n].FirstName}";
```

- Для добавления строк в циклах, особенно при работе с текстами больших размеров, используйте объект StringBuilder.

C#
```
var phrase = "lalalalalalalalalalalalalalalalalalalalalalalalalalalalala";
var manyPhrases = new StringBuilder();
for (var i = 0; i < 10000; i++)
{
    manyPhrases.Append(phrase);
}
//Console.WriteLine("tra" + manyPhrases);
```

## Неявно типизированные локальные переменные

- В случаях, когда тип переменной понятен из правой части назначения или когда точный тип не важен, рекомендуется использовать неявное типизирование для локальных переменных.

C#
```
var var1 = "This is clearly a string.";
var var2 = 27;
```

- Если тип в правой части назначения не является очевидным, не рекомендуется использовать var. Не полагайтесь на то, что имя метода предоставляет информацию о типе. Тип переменной можно считать очевидным, если используется оператор `new` или явное приведение типа.

C#
```
int var3 = Convert.ToInt32(Console.ReadLine());
int var4 = ExampleClass.ResultSoFar();
```

- Не полагайтесь на имя переменной при указании ее типа. Имя может быть неверным. В следующем примере имя переменной `inputInt` вводит в заблуждение. На самом деле это строка.

C#

```
var inputInt = Console.ReadLine();
Console.WriteLine(inputInt);
```

- Рекомендуется избегать использования `var` вместо dynamic. Используйте `dynamic`, если требуется определение типа во время выполнения.
- Используйте неявное ввод, чтобы определить тип переменной цикла в for циклах.

В следующем примере неявное типизирование используется в операторе `for`.

C#

```
var phrase = "lalalalalalalalalalalalalalalalalalalalalalalalalalalalalala";
var manyPhrases = new StringBuilder();
for (var i = 0; i < 10000; i++)
{
    manyPhrases.Append(phrase);
}
//Console.WriteLine("tra" + manyPhrases);
```

- Не используйте неявное ввод, чтобы определить тип переменной цикла в foreach циклах. В большинстве случаев тип элементов в коллекции не сразу очевиден. Имя коллекции не должно полагаться исключительно на определение типа его элементов.

В следующем примере явное типизирование используется в операторе `foreach`.

C#

```
foreach (char ch in laugh)
{
    if (ch == 'h')
        Console.Write("H");
    else
        Console.Write(ch);
}
Console.WriteLine();
```

**Примечание**

Следите за тем, чтобы случайно не изменить тип элемента итерируемой коллекции. Например, можно легко переключиться с **System.Linq.IQueryable** на **System.Collections.IEnumerable** в инструкции `foreach`, изменяющей выполнение запроса.

## Беззнаковые типы данных

Как правило, рекомендуется использовать `int` вместо беззнаковых типов. В C# обычно используется `int`. Использование `int` упрощает взаимодействие с другими библиотеками.

## Массивы

При инициализации массивов в строке объявления рекомендуется использовать сокращенный синтаксис. В следующем примере видно, что `var` нельзя использовать вместо `string[]`.

C#

```
string[] vowels1 = { "a", "e", "i", "o", "u" };
```

Если экземпляр создается явно, можно использовать `var`.

C#Копировать

```
var vowels2 = new string[] { "a", "e", "i", "o", "u" };
```

## Делегаты

Используйте Func<> и Action<> вместо определения типов делегатов. В классе определите метод делегата.

C#
```
public static Action<string> ActionExample1 = x => Console.WriteLine($"x is: {x}");

public static Action<string, string> ActionExample2 = (x, y) =>
    Console.WriteLine($"x is: {x}, y is {y}");

public static Func<string, int> FuncExample1 = x => Convert.ToInt32(x);

public static Func<int, int, int> FuncExample2 = (x, y) => x + y;
```

Вызывайте метод с помощью сигнатуры, которую определяет делегат Func<> или Action<>.

C#
```
ActionExample1("string for x");

ActionExample2("string for x", "string for y");

Console.WriteLine($"The value is {FuncExample1("1")}");

Console.WriteLine($"The sum is {FuncExample2(1, 2)}");
```

Если вы создаете экземпляры типа делегата, используйте сокращенный синтаксис. В классе определите тип делегата и метод с соответствующей сигнатурой.

C#
```
public delegate void Del(string message);

public static void DelMethod(string str)
{
    Console.WriteLine("DelMethod argument: {0}", str);
}
```

Создайте экземпляр типа делегата и вызовите его. В следующем объявлении используется сокращенный синтаксис.

C#
```
Del exampleDel2 = DelMethod;
exampleDel2("Hey");
```

В следующем объявлении используется полный синтаксис.

C#
```
Del exampleDel1 = new Del(DelMethod);
exampleDel1("Hey");
```

## Операторы try-catch и using при обработке исключений

- Рекомендуется использовать оператор try-catch для обработки большей части исключений.

C#
```
static string GetValueFromArray(string[] array, int index)
{
    try
    {
        return array[index];
    }
    catch (System.IndexOutOfRangeException ex)
```

```
        {
            Console.WriteLine("Index is out of range: {0}", index);
            throw;
        }
    }
```

- Использование оператора C# using упрощает код. При наличии оператора try-finally, код которого в блоке `finally` содержит только вызов метода Dispose, вместо него рекомендуется использовать оператор `using`.

В следующем примере оператор `try-finally` вызывает `Dispose` только в блоке `finally`.

C#
```
Font font1 = new Font("Arial", 10.0f);
try
{
    byte charset = font1.GdiCharSet;
}
finally
{
    if (font1 != null)
    {
        ((IDisposable)font1).Dispose();
    }
}
```

То же самое можно сделать с помощью оператора `using`.

C#
```
using (Font font2 = new Font("Arial", 10.0f))
{
    byte charset2 = font2.GdiCharSet;
}
```

Используйте новый using синтаксис , который не требует фигурных скобок:

C#Копировать
```
using Font font3 = new Font("Arial", 10.0f);
byte charset3 = font3.GdiCharSet;
```

## Операторы && и ||

Чтобы избежать возникновения исключений и увеличить производительность за счет пропуска необязательных сравнений, используйте && вместо & и || вместо | при выполнении сравнений, как показано в следующем примере.

C#
```
Console.Write("Enter a dividend: ");
int dividend = Convert.ToInt32(Console.ReadLine());

Console.Write("Enter a divisor: ");
int divisor = Convert.ToInt32(Console.ReadLine());

if ((divisor != 0) && (dividend / divisor > 0))
{
    Console.WriteLine("Quotient: {0}", dividend / divisor);
}
else
{
    Console.WriteLine("Attempted division by 0 ends up here.");
}
```

Если делитель равен нулю, второе условие в операторе `if` вызовет ошибку времени выполнения. && Но оператор замыкается, когда первое выражение имеет

значение false. Это означает, что второе выражение не будет вычисляться. Оператор & вычисляет оба, что приводит к ошибке во время выполнения, если `divisor` значение равно 0.

## Оператор new

- Используйте одну из сокращенных форм создания экземпляров объектов, как показано в следующих объявлениях. Во втором примере используется синтаксис, который появился в версии C# 9.

C#
```
var instance1 = new ExampleClass();
```
C#
```
ExampleClass instance2 = new();
```
Предыдущие объявления эквивалентны следующему объявлению.

C#
```
ExampleClass instance2 = new ExampleClass();
```
- Используйте инициализаторы объектов, чтобы упростить создание объектов, как показано в следующем примере.

C#
```
var instance3 = new ExampleClass { Name = "Desktop", ID = 37414,
    Location = "Redmond", Age = 2.3 };
```
В следующем примере задаются точно такие же свойства, как и в предыдущем, но без использования инициализаторов.

C#
```
var instance4 = new ExampleClass();
instance4.Name = "Desktop";
instance4.ID = 37414;
instance4.Location = "Redmond";
instance4.Age = 2.3;
```

## Обработка событий

Если вы определяете обработчик событий, который не нужно удалять позже, используйте лямбда-выражение.

C#
```
public Form2()
{
    this.Click += (s, e) =>
        {
            MessageBox.Show(
                ((MouseEventArgs)e).Location.ToString());
        };
}
```
Лямбда-выражение сокращает приведенное ниже традиционное определение.

C#
```
public Form1()
{
    this.Click += new EventHandler(Form1_Click);
}

void Form1_Click(object? sender, EventArgs e)
{
    MessageBox.Show(((MouseEventArgs)e).Location.ToString());
```

```
}
```

## Статические члены

Для вызова статических членов следует использовать имя класса: *ClassName.StaticMember*. В этом случае код становится более удобочитаемым за счет четкого доступа. Не присваивайте статическому элементу, определенному в базовом классе, имя производного класса. Во время компиляции кода его читаемость нарушается, и если добавить статический член с тем же именем в производный классе, код может быть поврежден.

## Запросы LINQ

- Используйте значимые имена для переменных запроса. В следующем примере используется `seattleCustomers` для клиентов, находящихся в Сиэтле.

```
C#
var seattleCustomers = from customer in customers
                       where customer.City == "Seattle"
                       select customer.Name;
```

- Рекомендуется использовать псевдонимы для уверенности в том, что в именах свойств анонимных типов верно используются прописные буквы при помощи правил использования прописных и строчных букв языка Pascal.

```
C#
var localDistributors =
    from customer in customers
    join distributor in distributors on customer.City equals distributor.City
    select new { Customer = customer, Distributor = distributor };
```

- Переименуйте свойства, если имена свойств в результате могут быть неоднозначными. Например, если запрос возвращает имя клиента и идентификатор распространителя, не оставляйте имена в виде `Name` и `ID`, а переименуйте их, чтобы было ясно, что `Name` — имя клиента и `ID` — идентификатор распространителя.

```
C#
var localDistributors2 =
    from customer in customers
    join distributor in distributors on customer.City equals distributor.City
    select new { CustomerName = customer.Name, DistributorID = distributor.ID };
```

- Рекомендуется использовать неявное типизирование в объявлении переменных запроса и переменных диапазона.

```
C#
var seattleCustomers = from customer in customers
                       where customer.City == "Seattle"
                       select customer.Name;
```

- Выравнивайте предложения запросов в соответствии с предложением from , как показано в предыдущих примерах.
- Используйте where предложения перед другими предложениями запросов, чтобы гарантировать, что более поздние предложения запросов работают с уменьшенным отфильтрованным набором данных.

```
C#
var seattleCustomers2 = from customer in customers
                        where customer.City == "Seattle"
                        orderby customer.Name
                        select customer;
```

- Используйте несколько `from` предложений вместо `join` предложения для доступа к внутренним коллекциям. Например, коллекция объектов `Student` может содержать коллекцию результатов тестирования. При выполнении следующего запроса возвращаются результаты, превышающие 90 балов, а также фамилии учащихся, получивших такие оценки.

```
C#
var scoreQuery = from student in students
                 from score in student.Scores!
                 where score > 90
                 select new { Last = student.LastName, score };
```