

## Задание №7.

### Автодокументирование кода

#### Цель задания:

*Получить навык комментирования кода с использованием автодокументирования языка C#.*

#### Комментирование кода:

В C#, как и в других Си-подобных языках, существуют два типа комментариев:

- 1) Однострочные комментарии `//`.
- 2) Многострочные комментарии `/* */`.

В первом варианте мы можем написать два символа `«//»` и весь текст после них до конца строки будет игнорироваться компилятором, например:

```
// Функция поиска максимума
public double FindMax(double[] values)
{
    double max = values[0];
    for (int i = 1; i < values.Length; i++)
    {
        if (values[i] > max)
        {
            max = values[i];
        }
    }

    return max; // Возвращаем максимальное число
}
```

Мы можем оставить поясняющий комментарий перед функцией или блоком кода внутри метода. Или оставить комментарий справа от строки с кодом (хотя требования к оформлению запрещают так делать).

Если комментарий занимает несколько строк, то разработчик должен оставлять в начале каждой строки два символа «//»:

```
// Функция поиска максимума.  
// Функция ожидает передачу массива, не равного null  
// и хранящего хотя бы один элемент.
```

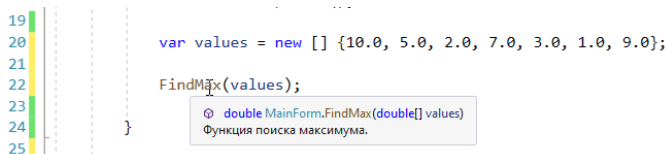
Другой подход — это использование многострочных комментариев, где начало комментария обозначается как «/\*», а конец — «\*/». Любой текст между ними воспринимается как комментарии, вне зависимости от переносов строк.

Однако оба данных подхода имеют существенный недостаток. Если вы оставите поясняющие комментарии к методам и полям класса, прочитать их можно только открыв исходный код класса. Однако в большинстве случаев комментарии содержат информацию об их правильном использовании, то есть они нужны в клиентском коде.

В C# одним из первых появилась концепция автодокументирования — комментариев к исходному коду, который отображается в клиентском коде непосредственно в точках использования. Такие комментарии имеют специальный формат, например:

```
/// <summary>  
/// Функция поиска максимума.  
/// </summary>  
public double FindMax(double[] values)  
{}
```

Благодаря такому комментарию, в точке вызова данного метода можно будет прочитать сопровождающий текст:



The screenshot shows a code editor with the following code:

```
19  
20 var values = new [] {10.0, 5.0, 2.0, 7.0, 3.0, 1.0, 9.0};  
21  
22 FindMax(values);  
23  
24  
25 }
```

A tooltip is displayed over the `FindMax` call on line 22, showing the signature `double MainForm.FindMax(double[] values)` and the summary text `Функция поиска максимума.`

Теперь стоит навести курсор мыши или ввести название метода в автодополнении, и всплывет всплывающая подсказка.

В отличие от обычных комментариев, автодокументирование позволяет писать исходный код, дающий подсказки разработчику во время его использования, а не только при чтении самого исходного кода.

Это особенно важно, так как разработчик далеко не всегда имеет доступ к исходному коду. Во-первых, вы можете использовать стороннюю библиотеку – обычные комментарии внутри кода сторонней библиотеки будут вам недоступны, в то время как комментарии автодокументирования будут отображаться даже для скомпилированных библиотек. Это в разы ускоряет работу со сторонними библиотеками, так как не надо отдельно штудировать документацию. Вы сможете читать описание классов и методов непосредственно в автодополнении.

Во-вторых, даже в рамках собственного исходного кода постоянное переключение между классами в разных вкладках потребует больше времени, чем прочтение комментария во всплывающей подсказке.

**Автодокументирование является обязательным стандартом промышленного кода.**

В случае языка C#, такие комментарии имеют специальный формат. В первую очередь, они начинаются `///`. Во-вторых, весь комментарий разделен на блоки, заключенные в так называемые xml-теги, обозначаемые угловыми скобками. Наиболее распространенный xml-тег – это тег `<summary>`. Каждый тег обязательно имеет открывающуюся и закрывающуюся часть. Закрывающуюся часть можно отличить по знаку `/` перед названием тега, например `</summary>`. Все строчки между тегами содержат полезную информацию и будут отображаться во всплывающих подсказках.

Теги нужны среде разработки для распознавания комментариев как автодокументации. Если допустить ошибку в названии тега, не поставить закрывающийся тег или не поставить в начале знак `///`, среда разработки воспримет текст как обычный комментарий и не будет отображать его в подсказках.

Основные теги:

- `<summary>` - «Описание», используется для любых элементов исходного кода, классов, перечислений, полей, свойств, методов, конструкторов.
- `<param>` - «Параметр», используется для описания входных аргументов метода, если они есть.
- `<returns>` - «Возвращает», используется для описания выходных данных метода, если они есть.

- `<see>` - «Смотреть», позволяет добавить гиперссылку внутри комментария для быстрой навигации по исходному коду.

Так как комментарии используют xml-теги, автодокументирование кода в C# также часто называют xml-комментариями.

### Комментарии к классам и другим типам данных

К классам, структурам или перечислениям указывается тег `<summary>`, описывающий назначение класса.

```
/// <summary>
/// Хранит данные о человеке и его контактных данных.
/// </summary>
public class Person
```

Текст внутри `<summary>` должен отвечать на вопрос, что делает класс, например, «хранит данные», «предоставляет методы за сохранение и загрузку данных в файл», «реализует статистическую обработку данных», «предоставляет типовые шаблоны поисковых запросов в БД» и т.д. Укажите одну наиболее существенную функцию класса в одно-два предложения.

### Комментарии к полям

Комментарии к любым полям также состоят из одного тега `<summary>`:

```
/// <summary>
/// Уникальный идентификатор для всех объектов данного класса.
/// </summary>
private readonly int _id;
```

Комментарий должен кратко ответить на вопрос, что это за данные.

### Комментарии к свойствам

Комментарий также состоит только из тега `<summary>`. Сообщает о том, к каким данным он предоставляет доступ. Если есть важная информация об ограничениях на входные данные, её стоит описать в комментарии:

```
/// <summary>
/// Возвращает и задает фамилию человека. Должна состоять только из букв.
/// </summary>
```

```

public string Surname
{
    get
    {
        return _surname;
    }
    set
    {
        Validator.AssertStringOnLetters(value,
nameof(Surname));
        _surname = value;
    }
}

```

К важной информации можно отнести требования к валидации, может ли входной аргумент принимать значение null. Если речь идет о коллекции (свойство предоставляет доступ к массиву или списку), можно указать ограничения на количество элементов, например, «массив должен содержать строго 4 значения».

Текст внутри свойства имеет специальный шаблон:

- Если свойство имеет только get, то свойство начинается с фразы «Возвращает ...».
- Если свойство имеет get и set, то текст начинается с фразы «Возвращает и задаёт».

Такое простое правило при составлении текста помогает быстро отличить обычные свойства от свойств, доступных только на чтение. Обязательно к соблюдению.

### Комментарии к методам

Наиболее объёмная часть комментариев внутри любого класса, так как для метода следует описывать блоки <summary>, каждый входной аргумент и выходные данные.

```

/// <summary>
/// Проверяет, что строка состоит только из букв.
/// </summary>
/// <param name="value">Проверяемая строка.</param>
/// <param name="propertyName">Имя свойства или объекта, которое
/// подлежит проверке.</param>
public static void AssertStringOnLetters(

```

```

        string value, string propertyName)
    {
        for (int i = 0; i < value.Length; i++)
        {
            if (!char.IsLetter(value[i]))
            {
                throw new
                    ArgumentException(
                        $"{propertyName} must contains letters only");
            }
        }
    }

    /// <summary>
    /// Проверяет, что строка состоит только из букв.
    /// </summary>
    /// <param name="value">Проверяемая строка.</param>
    /// <returns>Возвращает true, если строка состоит из букв.
    /// И false, если есть хотя бы одна не буква.</returns>
    public static bool IsNameOnLetters(string value)
    {
        for (int i = 0; i < value.Length; i++)
        {
            if (!char.IsLetter(value[i]))
            {
                return false;
            }
        }

        return true;
    }
}

```

Блок param прописывается для каждого входного аргумента, причем в качестве поля name=«» должно указываться название параметра ровно так, как оно указано в объявлении самого метода.

Заполнение такого количества комментариев можно показаться бесполезной работой и первое время будет занимать много сил. Однако понимание важности xml-комментариев приходит в тот момент, когда разработчик в первый раз столкнется с использованием сторонней библиотеки без какой-либо документации. В таких случаях приходится наугад вызывать методы из библиотеки,

надеясь получить нужный результат – на это могут уйти долгие часы. Написание xml-комментариев для готового класса может занять пять-десять минут, однако их наличие может сэкономить десятки часов при работе с классами, а также избавить от нелепых ошибок при вызове методов. Например, когда вы не знали, что метод не может принимать значение null в качестве аргумента, и вы должны были реализовать эту проверку самостоятельно.

Разработчики сред разработки, таких как Visual Studio и Rider упростили вам задачу. Вам достаточно написать `///` перед объявлением готового метода, и среда разработки сама создаст вам структуру всего комментария – вам останется только написать содержательную часть комментария.

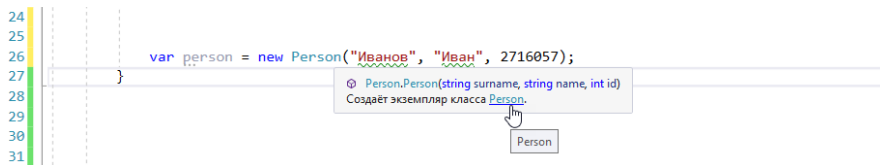
### Комментарии к конструкторам

Конструкторы являются разновидностью методов, поэтому структура их комментариев также состоит из `<summary>` и `<param>`. Однако текст внутри комментария формируется по определенному шаблону:

```
/// <summary>
/// Создаёт экземпляр класса <see cref="Person"/>.
/// </summary>
/// <param name="surname">Фамилия. Должна состоять только из
букв.</param>
/// <param name="name">Имя. Должно состоять только из
букв.</param>
/// <param name="id">Уникальный идентификатор.
/// Класс не контролирует уникальность присвоенного id.</param>
public Person(string surname, string name, int id)
{
    Surname = surname;
    Name = name;
    _id = id;
}
```

Здесь в качестве шаблона выступает фраза «Создаёт экземпляр класса», после которой идет тег `<see>` с ссылкой на класс, внутри которого создан конструктор (например, ссылка на класс `Person` в комментарии конструктора `Person` выше).

Как было сказано ранее, тег `<see>` позволяет создать внутри комментария гиперссылку, которая будет работать внутри всплывающего окна:



Такие ссылки значительно упрощают навигацию по классам в клиентском коде, особенно при чтении чужого кода (кода, написанного вашим коллегой в команде).

## Общие правила и стратегии написания html-комментариев

Есть несколько дополнительных правил, которые следует помнить:

- Комментарии следует заканчивать точкой, как обычные предложения. Это особенно важно, когда на основе html-комментариев собирается отдельный pdf или html-документ, документация для сторонних разработчиков. Без точек в конце предложения весь текст сольётся в одно большое предложение.
- Грамматические ошибки должны исправляться сразу же при обнаружении. Ошибки в комментариях не так страшны, как ошибки в названиях классов и методов, но их также следует исправлять.
- Внутри методов используются обычные однострочные комментарии. Не надо писать блоки `<summary>` внутри методов, конструкторов и свойств.
- Если реализация класса или метода основана на готовом решении, взятом из интернета или из книги – оставьте ссылку на первоисточник внутри комментария. Разработчики часто копируют код со Stackoverflow, однако помимо кода, в первоисточнике хранится объяснение, почему это сделано именно так. Чтобы в будущем не забыть, почему вы решили скопировать готовое решение, и какие нюансы есть в его использовании, оставьте ссылку!

```
///Класс реализует динамическую привязку данных. Идея подсмотрена  
здесь: https://stackoverflow.com/questions/13083126/dynamic-binding-to-c-sharp-events-using-reflection  
///  
///</summary>
```

- Если это ссылка на книжку, то укажите её автора и название, а также страницу или главу, где было взято решение.



Есть разные стратегии к написанию xhtml-комментариев. Большинство стандартов оформления кода указывают, что xhtml-комментарии должны обязательно прописываться у всех открытых членов класса (любые свойства и методы с модификатором `public`). Однако работая с legacy-кодом, понимаешь, что комментарии важно оставлять не только к открытым, но и к закрытым членам класса, так как они также могут иметь ограничения, особенности реализации и неочевидные причины алгоритмических и архитектурных решений. Любое нетривиальное решение, любое ограничение на входные или хранимые данные, любое магическое число – всё должно иметь поясняющий комментарий.

Кроме того, в сети или литературе можно найти совет «Хорошо написанный исходный код не требует комментариев». Не льстите себе. Понимание того, что такое хорошо написанный код, может прийти только после долгих лет практики в разных проектах и разных командах разработки. До тех пор, вы обязаны писать комментарии к вашему коду, насколько бы простым и очевидным он вам не казался.

В рамках заданий, в качестве дополнительной практики по написанию комментариев, принимается стратегия комментирования всех членов класса – полей, свойств, методов, конструкторов, вне зависимости от их модификатора доступа. Также комментарии должны стоять у самого класса, структуры или перечисления. Допускается не писать xhtml-комментарии у обработчиков событий внутри форм или пользовательских элементов управления.

#### **Последовательность выполнения:**

1. Написать xhtml-комментарии для всего исходного кода, который был написан в рамках предыдущих заданий – все классы, все перечисления, все поля, методы, свойства и конструкторы.
2. Проверьте правильность комментариев и их полноту. Проверьте соответствие текста в комментариях шаблонам, которые представлены в разделе выше.
3. Проверьте комментарии на наличие грамматических ошибок и точек в конце предложений. Убедитесь, что после написания xhtml-комментариев между членами класса всё равно осталась одна пустая строка. Убедитесь, что после расстановки xhtml-комментариев не «съехала» табуляция в исходном коде.