



# ООП в С#

(методы, параметры, конструкторы)

Андрей Голяков

# Организация кода

---

На примере домашней работы стало заметно, что файл программы становится достаточно **длинным** и постоянно скролить экран для перехода между различными областями становится **неудобно**.

Обратите внимание, что изначальный класс **Program**, содержащий основной поток нашей программы в методе Main располагается в файле с именем **Program.cs**.

Это **хорошая практика** — иметь отдельный файл для каждого класса программы с именем, совпадающим с названием класса.

На примере рассмотрения домашнего задания разносим классы **Program** и **Person** по разным файлам: **Program.cs** и **Person.cs**.



# Методы

---

**Метод** — это определенный **набор инструкций**, который можно многократно использовать. Иногда используют термин **подпрограмма**. Методы объекта, с точки зрения инкапсуляции, помогают решать задачи объекта или задачи пользователя над объектом.

Например, посмотрим на наш класс Program, знакомый нам с первого урока. Там есть **метод Main**, который является точкой входа в нашу программу. Этот метод содержит основной поток выполнения кода программы.



# Возвращаемые значения

---

Метод **может возвращать значение** любого типа данных.

Например, вспомним класс `Random`, которым мы пользовались, чтобы получить произвольное число.

```
// создаем экземпляр класса
Random rand = new Random();

// вызываем метод Next()
// он возвращает значение типа int
// которое мы сохраняем в переменной r
int r = rand.Next();
```

От нас скрыто, что именно происходит внутри этого метода, и, в данном случае, нас это не интересует. Мы хотим лишь использовать метод для получения значения.



# Пример имплементации метода

Посмотрим на наш класс Person из домашнего задания, вот его свойство AgeInFourYears:

```
public int AgeInFourYears
{
    get { return Age + 4; }
}
```

Напишем метод со схожим функционалом:

```
public int AgeInFourYearsMethod()
{
    return Age + 4;
}
```

Обратите внимание на ключевое слово **return** – оно выполняет 2 функции:

1. **Определяет значение**, которое будет возвращаться
2. Возвращает его, **завершая** при этом исполнение метода.

С точки зрения программы это логично, если весь метод писался для того, чтобы рассчитать и вернуть значение, как только мы готовы это сделать – нет смысла выполнять дальше какой-либо код метода.

# Работа с методами класса

Свойство `PropertiesString` также должно быть модифицировано, так как **при вызове метода**, чтобы отличать его от свойств и полей, **необходимо указывать круглые скобки**:

```
// Вызов свойства AgeInFourYears
public string PropertiesString
{
    get { return $"Name: {Name}, age in 4 years: {AgeInFourYears}."; }
}
```

```
// Вызов метода AgeInFourYearsMethod()
public string PropertiesString
{
    get { return $"Name: {Name}, age in 4 years: {AgeInFourYearsMethod()}."; }
}
```



# Отсутствие значения метода: **void**

Если метод не должен ничего возвращать, например, когда он просто содержит ряд действий по вводу/выводу, ему необходимо указывать вместо типа данных `void`.

Это не специальный тип данных, **это ключевое слово, означающее “пусто”**, т.е. вы не забыли указать тип, а явно сказали, что значения в результате выполнения метода возвращено не будет.

```
// Пример метода, который не возвращает значение
public void WriteDescription()
{
    Console.WriteLine(Description);
}
```



# Параметры метода

Параметры позволяют **передать в метод** некоторые входные данные.

```
// Параметр yearsToAdd позволяет передать
// произвольное количество лет, что
// делает наш метод более общим
public int AgeInSomeYears(int yearsToAdd)
{
    return Age + yearsToAdd;
}
```

```
// Теперь при вызове метода я обязан указывать
// значение этого параметра, так как оно
// является необходимым для логики работы моего метода
public string PropertiesString
{
    get { return $"Name: {Name}, age in 4 years: {AgeInSomeYears(4)}."; }
}
```





# Опциональные параметры

С версии 4.0 C# поддерживает опциональные параметры.

Это позволяет определить **используемое по умолчанию значение** для параметра метода. Данное значение будет использоваться в том случае, если для параметра не указан соответствующий аргумент при вызове метода.

```
// Теперь наш параметр years - опциональный
public void WriteProperties(int years = 10)
{
    Console.WriteLine(
        $"Name: {Name}, age in {years} years: {AgeInSomeYears(years)}.");
}

// Теперь при вызове этого метода
// значение можно не указывать
for (int i = 0; i < persons.Length; i++)
{
    persons[i].WriteProperties();
    persons[i].WriteProperties(12);
}
```



# Перегрузка методов

Иногда возникает необходимость создать **один и тот же метод с разным набором параметров** и в зависимости от имеющихся параметров применять определенную версию метода. Такая возможность еще называется **перегрузкой методов**:

```
// Перегруженный метод UpdateProperties
public void UpdateProperties(string name, int age)
{
    Name = name;
    Age = age;
}
```

```
public void UpdateProperties(int age)
{
    Age = age;
}
```

```
// При вызове будет выбран метод, подходящий по аргументам
person.UpdateProperties("Andrei Golyakov", 36);
person.UpdateProperties(36);
```



# Конструкторы

Заметьте, что конструктор очень похож на обычный метод, но у него **два важных отличия**:

- **Имя** всегда совпадает с именем класса
- **Не указан тип данных** значения, которое должно вернуться.

После объявления явного конструктора, конструктор по умолчанию больше не работает! Можно это исправить объявив явно конструктор по умолчанию без параметров:

```
public Person() { }
```

Используем наш конструктор с параметрами:

```
Console.Write($"Enter name: ");  
var name = Console.ReadLine();  
  
Console.Write($"Enter age: ");  
var age = int.Parse(Console.ReadLine());  
  
var person = new Person(name, age);
```



# Конструкторы

Для инициализации объектов часто используют специальные методы — конструкторы.

Эти методы устанавливают значения полей объекта, а также могут производить некоторые операции по подготовке, при создании объекта.

В каждом классе уже реализован конструктор по умолчанию без параметров. Именно его мы вызываем, когда пишем

```
var persons = new Person();
```

Вот это `new Person()` и есть конструктор, создающий объект.

Мы можем перегрузить его дефолтную имплементацию:

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



# Partial classes

---

В случае, если код внутри файла класса становится слишком объемным, можно выделить логические части этого класса и разделить его на несколько файлов.

Чтобы компилятор правильно относился к тому, что объявление класса происходит в нескольких файлах сразу, используют ключевое слово `partial`.

При разделении очень важно не просто скопировать половину кода в новый файл, а выделить некую логику такого разделения.

В нашем случае пример будет немного наигран, так как при таком размере, в действительности, нет необходимости делить класс на несколько файлов. Однако, допустим, я хочу выделить всю логику вывода на экран в отдельный файл, чтобы он не мешался при работе с основной логикой класса:



# Partial classes

```
// Файл Person.cs
public partial class Person
{
    public string Name { get; set; }
    public int Age { get; set; }

    public Person() { }

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

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

    public void UpdateProperties(int age)
    {
        Age = age;
    }

    public void UpdateProperties(string name)
    {
        Name = name;
    }
}
```

```
// Файл Person.Output.cs
public partial class Person
{
    public void WriteProperties(int years)
    {
        Console.WriteLine(GetPropertiesString(years));
    }

    private int AgeInSomeYears(int yearsToAdd)
    {
        return Age + yearsToAdd;
    }

    private string GetPropertiesString(int years)
    {
        return $"Name: {Name}, " +
            $"age in {years} years: " +
            $"{AgeInSomeYears(years)}.";
    }
}
```



# Домашнее задание

Написать класс одной записи будильника **ReminderItem** (как будильник в телефоне), который будет иметь

- Свойства:
  - **AlarmDate** типа DateTimeOffset (дата/время будильника)
  - **AlarmMessage** типа string (сообщение, соответствующее будильнику)
  - **TimeToAlarm** типа TimeSpan (время до срабатывания будильника), должно быть read-only, рассчитываться как текущее время минус AlarmDate
  - **IsOutdated** типа bool (просрочено ли событие), должно быть read-only, рассчитываться как
    - true, если TimeToAlarm больше либо равно 0
    - false, если TimeToAlarm меньше 0
- Методы:
  - **Конструктор**, который будет инициализировать значения AlarmDate и AlarmMessage.
  - **WriteProperties()**, который будет выводить на экран все свойства экземпляра класса в формате “Имя поля : значение”.

В основном потоке программы создать два экземпляра класса ReminderItem и вывести их параметры на экран.



# Спасибо за внимание.

