Темы урока

Организация кода (отдельные файлы классов)

- 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**.

Самостоятельная работа

Разнести по разным файлам классы **Program** и **Pet**, с которыми мы работали на прошлом уроке.

Методы

Мы уже употребляли этот термин несколько раз, давайте разберемся в точности, что же он означает?

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 входу 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 1

Давайте посмотрим на наш класс Person из домашнего задания:

```
class Person
{
	public string Name { get; set; }

	public int Age { get; set; }

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

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

Обратите внимание, что очень похожим на метод является свойство AgeInFourYears. В действительности, это свойство может быть легко представлено методом – необходимо лишь **добавить круглые скобки** и **убрать обертку get { }**, оставив его содержимое:

```
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.

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

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

Пример 2 (посложнее)

Давайте вернемся к нашему классу Pet из самостоятельной работы. Усложним его немного. Пусть теперь в классе хранится не возраст в годах, а дата рождения питомца. Удаляем поле Age, добавляем поле DateOfBirth типа DateTimeOffset. Теперь список полей будет выглядеть так:

```
public string Kind;
public string Name;
public char Sex;
public DateTimeOffset DateOfBirth;
```

Далее, зная дату рождения, мы всегда можем определить текущий возраст питомца в днях. При этом информация в нашем объекте не будет устаревать со временем. Если бы мы запустили старый код через год, мы получили бы прошлогодние данные. Теперь возраст в днях всегда будет актуальный.

Пара слов о типе **DateTimeOffset**. Это достаточно молодой тип данных для хранения даты/времени с учетом часового пояса. Мы сейчас не будем очень подробно останавливаться на этом классе, нам понадобится метод **Subtract** для “вычитания” одной даты из другой с целью получения временного интервала между ними. Для работы с временными интервалами используется класс **TimeSpan**. Он позволяет представить временной интервал в любой размерности - от миллисекунд до дней.

Напишем такой метод Age:

```
public int Age()
{
	TimeSpan age = DateTimeOffset.UtcNow.Subtract(DateOfBirth);
	return Convert.ToInt32(Math.Floor(age.TotalDays / 365.242));
}
```

Сначала мы делим количество дней на 365.242 это честное количество дней в “среднем” году, учитывая високосные. Затем, поскольку нас интересует количество полных лет, мы оставляем только целую часть с помощью Math.Floor. Не смотря на то, что мы знаем, что число там целое, тип данных все еще не совпадает с нашим int, объявленным в заголовке метода. Необходимо воспользоваться одним из способов приведения типов. Я воспользуюсь Convert.ToInt32.

Далее необходимо обновить код внутри свойства Description: при вызове Age необходимо добавить скобки, так как теперь это не поле, а метод:

```
public string Description {
	get { return $"{Name} is a {Kind} ({Sex}) of {Age()} years old."; }
}
```

Теперь осталось поправить код в основной программе. Вместо указания Age нужно указывать DateOfBirth:

```
pet1.DateOfBirth = DateTimeOffset.Parse("2011-03-14");
```

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.

Самостоятельная работа

Добавьте метод WriteDescription, который бы выводил описание нашего питомца на экран с помощью Console.WriteLine.

Решение:

```
// Pet.cs
public void WriteDescription()
{
	Console.WriteLine(Description);
}
```

```
// Program.cs

// Вместо вызова Console.WriteLine(pet1.Description); вызываем новый метод

pet1.WriteDescription();
```

Параметры метода

Параметры позволяют передать в метод некоторые входные данные.

Пример

Давайте посмотрим на наш метод AgeInFourYearsMethod класса Person.
Он очень специфичные, хотя мы могли бы сделать его более общим с помощью параметров – величин которые могут подаваться на вход методу.

Для этого нашу константу 4 надо сделать переменной величиной внутри метода. Я назвал ее yearsToAdd, и объявил ее с таким же типом данных, как и само свойство Age.

Теперь я могу использовать эту переменную внутри тела функции как обычную переменную.

Также, я переименую сам метод, так как он уже будет рассчитывать возраст не только через года, а через произвольное количество лет.

```
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)}."; }
}
```

Давайте по аналогии с нашим классом Pet избавимся от необходимости каждый раз выводить параметры используя свойство PropertiesString, вместо него мы напишем метод WriteProperties:

```
public void WriteProperties()
{
	Console.WriteLine($"Name: {Name}, age in 4 years: {AgeInSomeYears(4)}.");
}
```

Мы видим, что метод весьма специфичен, давайте сделаем его более общим. Также как и в методе AgeInSomeYears добавим переменную типа int и будем выводить на экран возраст через любое количество лет:

```
public void WriteProperties(int years)
{
	Console.WriteLine(
		$"Name: {Name}, age in {years} years: {AgeInSomeYears(years)}.");
}
```

Я специально не использовал то же самое имя, чтобы показать, что имена параметров не обязательно должны быть одинаковые.

Теперь исправим код основной программы, нам уже не нужно писать каждый раз Console.WriteLine, так как метод WriteProperties делает это за нас. Зато мы теперь обязаны передавать определенное количество лет. Давайте вернем сюда нашу 4. Однако теперь наш класс способен решить задачу в общем виде.

```
for (int i = 0; i < persons.Length; i++)
{
	persons[i].WriteProperties(4);
	persons[i].WriteProperties(12);
}
```

Самостоятельная работа

Добавить к классу Pet еще одно свойство ShortDescription, которое будет возвращать только имя и вид животного (без пола и возраста).

Затем добавить в метод WriteDescription параметр булева типа showFullDescrption, а в сам метод добавить логику выбора нужного свойства в зависимости от параметра функции.

В изменить основную логику программы таким образом, чтобы в первом случае (для pet1) выводилось короткое описание, а во втором (для pet2) — полное описание.

Пример выполнения

```
// Pet.cs
public string Description {
	get { return $"{Name} is a {Kind} ({Sex}) of {Age()} years old."; }
}

public string ShortDescription
{
	get { return $"{Name} is a {Kind}."; }
}

public void WriteDescription(bool showFullDescrption)
{
	Console.WriteLine(Description);
}
```

```
// Program.cs
using System;

class Program
{
	static void Main(string[] args)
	{
		Pet pet1 = new Pet();
		pet1.Kind = "Cat";
		pet1.Name = "Tom";
		pet1.Sex = 'M';
		pet1.DateOfBirth = DateTimeOffset.Parse("2011-03-14");
		pet1.WriteDescription(false);

		Pet pet2 = new Pet
		{
			Kind = "Mouse",
			Name = "Minnie",
			Sex = 'F',
			DateOfBirth = DateTimeOffset.Parse("2017-03-14")
		};
		pet1.WriteDescription(true);
	}
}
```

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.

Пример

Давайте сделаем так, чтобы в классе Person наш метод WriteProperties, требующий обязательного указания количества лет, чтобы рассчитать возраст в будущем, считал, что, необходимо выводить возраст через 10 лет, если значение не указано.

Все, что для этого надо сделать, после объявления параметра написать его значение по умолчанию через знак равенства:

```
// Person.cs

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

Теперь в основном коде мы можем вызывать метод без параметров, в таком случае он будет вести себя так, как будто ему передано значение 10:

```
// Program.cs

// Writing out the results
for (int i = 0; i < persons.Length; i++)
{
	persons[i].WriteProperties();
	persons[i].WriteProperties(12);
}
```

Самостоятельная работа

В классе Pet сделайте метод параметр showFullDescrption метода WriteDescription опциональным со значением по умолчанию false.

В основном коде уберите явную передачу значения false при вызове WriteDescription.

Решение:

```
// Pet.cs
public void WriteDescription(bool showFullDescrption = false)
{
	Console.WriteLine(Description);
}
```

```
// Program.cs
...
pet1.WriteDescription();
```

Перегрузка методов

Иногда возникает необходимость создать один и тот же метод, но с разным набором параметров. И в зависимости от имеющихся параметров применять определенную версию метода. Такая возможность еще называется перегрузкой методов (method overloading).

Пример

Добавим в класс Person метод UpdateProperties, который будет обновлять свойства Name и Age через переданные параметры:

```
// Person.cs

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

Можно перегрузить этот метод для случая, если мы хотим поменять только один из параметров:

```
// Person.cs

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

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

Теперь при использовании, когда мы напишем вызов метода UpdateProperties мы сможем выбрать одну из трех перегрузок, указывая конкретный набор параметров. .NET поймет по количеству и типу передаваемых значений, какой именно метод мы имели в виду:

```
// Program.cs

persons[0].UpdateProperties("Andrei Golyakov");
persons[0].UpdateProperties("Andrei Golyakov", 36);
persons[0].UpdateProperties(36);
```

Самостоятельная работа

Создайте перегруженные методы UpdateProperties.
Один из них должен обновлять только один параметр: Name.
Второй – все 4 параметра: Kind, Name, Sex и DateOfBirth.

В основном потоке программы после вывода первоначальных параметров питомцев, вызовите для животного pet1 первый метод, для pet2 – второй и выведите на экран измененные параметры.

Решение:

```
// Pet.cs

public void UpdateParameters(string name)
{
	Name = name;
}

public void UpdateParameters(
	string kind,
	string name,
	char sex,
	DateTimeOffset dateOfBirth)
{
	Kind = kind;
	Name = name;
	Sex = sex;
	DateOfBirth = dateOfBirth;
}
```

```
// Program.cs
...
pet1.UpdateParameters("Garfield");
pet1.WriteDescription();

pet1.UpdateParameters(
	"Mouse",
	"Mickey",
	'M',
	DateTimeOffset.Parse("2017-01-01"));
pet1.WriteDescription(true);
```

Конструкторы

Для инициализации объектов часто используют специальные перегруженные методы, которые являются конструкторами. Эти методы очень похожи на наши, только что написанные, методы Update, только они не устанавливают значения при создании объекта, а не после этого.

Пример

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;
}
```

Заметьте, что очень похоже 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, посмотрите на Program.cs, наше объявление persons[i] = new Person(); больше не является приемлемым.

Можно это исправить объявив явно конструктор по умолчанию без параметров:

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

Это вернет работоспособность нашему коду.

Однако, давайте используем наш конструктор с параметрами:

```
Console.Write($"Enter name {i}: ");
var name = Console.ReadLine();

Console.Write($"Enter age {i}: ");
var age = int.Parse(Console.ReadLine());

persons[i] = new Person(name, age);
```

Самостоятельная работа

Напишите конструктор с параметрами для нашего класса Pet, где бы устанавливались все четыре параметра.
Обновите код программы, создающей экземпляры класса таким образом, чтобы они использовали новый конструктор.

Пример:

```
// Pet.cs

public Pet(string kind, string name, char sex, DateTimeOffset dateOfBirth)
{
	Kind = kind;
	Name = name;
	Sex = sex;
	DateOfBirth = dateOfBirth;
}
```

```
// Program.cs

Pet pet1 = new Pet(
	"Cat",
	"Tom",
	'M',
	DateTimeOffset.Parse("2011-03-14"));
pet1.WriteDescription();

Pet pet2 = new Pet(
	"Mouse",
	"Minnie",
	'F',
	DateTimeOffset.Parse("2017-03-14"));
pet2.WriteDescription(true);
```

Partial классы

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**.

```
public partial class Person
```

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:

```
// 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 меньше 0
        - true, если TimeToAlarm меньше 0

- **Конструктор**, который будет инициализировать значения AlarmDate и AlarmMessage.

- Метод **WriteProperties()**, который будет выводить на экран все свойства экземпляра класса в формате “Имя поля : значение”.

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