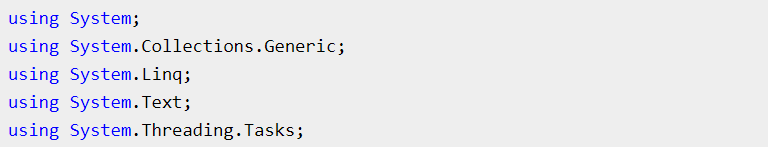
Hello world explained

In the previous chapter, we tried writing a piece of text to the console, in our first C# application. To see some actual progress, we didn't go into much detail about the lines of code we used, so this chapter is an explanation of the Hello world example code. As you can probably see from the code, some of the lines look similar, so we will bring them back in groups for an individual explanation. Let's start with the shortest and most common characters in our code: The { and }. They are often referred to as curly braces, and in C#, they mark the beginning and end of a logical block of code. The curly braces are used in lots of other languages, including C++, Java, JavaScript and many others. As you can see in the code, they are used to wrap several lines of code which belong together. In later examples, it will be clearer how they are used.

Now let's start from the beginning:



**using** is a keyword, highlighted with blue by the editor. The using keyword imports a namespace, and a namespace is a collection of classes. Classes brings us some sort of functionality, and when working with an advanced IDE like Visual Studio, it will usually create parts of the trivial code for us. In this case, it created a class for us, and imported the namespaces which are required or expected to be used commonly. In this case, 5 namespaces are imported for us, each containing lots of useful classes. For instance, we use the Console class, which is a part of the System namespace.

On the other hand, we don't really use the System.Linq namespace yet (for example), so if you're a purist, you may choose to remove this line, but it won't make much of a difference at this point.

As you can see, we even get our own namespace:



The namespace ConsoleApp1 is now the main namespace for this application, and new classes will be a part of it by default. Obviously, you can change this, and create classes in another namespace. In that case, you will have to import this new namespace to use it in your application, with the using statement, like any other namespace.

Next, we define our class. Since C# is truly an Object Oriented language, every line of code that actually does something, is wrapped inside a class. In this case, the class is simply called Program:



We can have more classes, even in the same file. For now, we only need one class. A class can contain several variables, properties and methods, concepts we will go deeper into later on. For now, all you need to know is that our current class only contains one method and nothing else. It's declared like this:



This line is probably the most complicated one in this example, so let's split it up a bit. The first word is **static**. The static keyword tells us that this method should be accesible without instantiating the class, but more about this in our chapter about classes.

The next keyword is **void**, and tells us what this method should return. For instance, it could be an integer or a string of text, but in this case, we don't want our method to return anything (C# uses the keyword void to express the concept of nothing).

The next word is **Main**, which is simply the name of our method. This method is the so-called entry-point of our application, that is, the first piece of code to be executed, and in our example, the only piece to be executed.

Now, after the name of a method, a set of arguments can be specified within a set of parentheses. In our example, our method takes only one argument, called **args**. The type of the argument is a **string**, or to be more precise, an array of strings, but more on that later. If you think about it, this makes perfect sense, since Windows applications can always be called with an optional set of arguments. These arguments will be passed as text strings to our main method.

And that's it. You should now have a basic understanding of our first C# application, as well as the basic principles of what makes a console application work.

# Variables

A variable can be compared to a storage room, and is essential for the programmer. In C#, a variable is declared like this:

<data type> <name>;

An example could look like this:

string name;

That's the most basic version, but the variable doesn't yet have a value. You can assign one at a later point or at the same time as declaring it, like this:



If this variable is not local to the method you're currently working in (e.g. a class member variable), you might want to assign a visibility to the variable:

<visibility> <data type> <name> = <value>;

And a complete example:



The visibility part is related to classes, so you can find a more complete explanation of it in the chapter about classes. Let's concentrate on the variable part with an example where we actually use a couple of them:

// See variables example 1

Okay, a lot of this has already been explained, so we will jump directly to the interesting part. First of all, we declare a couple of variables of the string type. A string simply contains text, as you can see, since we give them a value straight away. Next, we output a line of text to the console, where we use the two variables. The string is made up by using the + characters to "collect" the different parts.

Next, we ask the user to enter a new first name, and then we use the ReadLine() method to read the user input from the console and enter it into the firstName variable. Once the user presses the Enter key, the new first name is assigned to the variable, and in the next line we output the name presentation again, to show the change. We have just used our first variable and the single most important feature of a variable: The ability to change its value at runtime.

Another interesting example is doing math. Here is one, based on a lot of the same code we have just used:

// See variables example 2

Put this in our Main method, and try it out. The only new "trick" we use here, is the int.Parse() method. It simply reads a string and converts it into an integer. As you can see, this application makes no effort to validate the user input, and if you enter something which is not a number, an exception will be raised. More about those later.

## **Variables & scope**

So far, we have only used local variables, which are variables defined and used within the same method. In C#, a variable defined inside a method can't be used outside of this method - that's why it's called local. If you're familiar with other programming languages, you may also know about global variables, which can be accessed from more places, but C# doesn't support the concept of global variables. Instead, you can define a field on a class, which can be accessed from all the methods of this class. Allow me to demonstrate this with an example:

// See variables example 3

Notice the **helloClass** member, declared on the class scope instead of inside a method - this will allow us to access it from both our Main() method as well as our own DoStuff() method. That is not true for our **helloLocal** variable, which has been declared inside the Main() method and can therefore only be used inside of this specific method.

The concept of differentiating between where a variable has been declared is called **scoping** and it prevents your code from becoming a huge mess of variables which can be changed from too many places. Another technique that helps us with this is called member visibility (in this case illustrated with the private keyword), which we'll discuss in the chapter about classes.

## **Summary**

Variables allow you to store data of various types, e.g. text strings, numbers or custom objects. There are local variables, which are accessible inside of the method in which it has been defined, and then there are class fields, which can be accessed from all the methods of the class and even outside of the class, if the visibility permits it.

# Data types

Data types are used everywhere in a programming language like C#. Because it's a strongly typed language, you are required to inform the compiler about which data types you wish to use every time you declare a variable, as you will see in the chapter about variables. In this chapter we will take a look at some of the most used data types and how they work.

**bool** is one of the simplest data types. It can contain only 2 values - false or true. The bool type is important to understand when using logical operators like the if statement.

**int** is short for integer, a data type for storing numbers without decimals. When working with numbers, int is the most commonly used data type. Integers have several data types within C#, depending on the size of the number they are supposed to store.

**string** is used for storing text, that is, a number of chars. In C#, strings are immutable, which means that strings are never changed after they have been created. When using methods which changes a string, the actual string is not changed - a new string is returned instead.

**char** is used for storing a single character.

**float** is one of the data types used to store numbers which can contain decimals.

## **Summary**

These are just the most basic data types in C# and I only told you the very basic stuff about them, because it's a pretty dry subject and you may want to see some data types in action before you read more about them. So, move on to the next article, where we'll be using variables to contain data of various types!

Later in this tutorial, we'll go much more in depth with all the various data types found in the .NET framework.

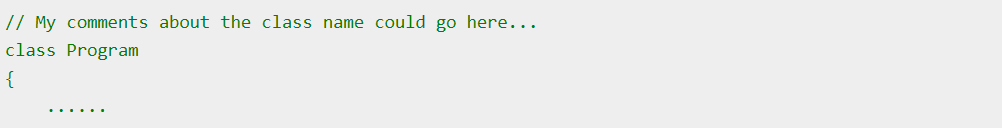
# Code Comments

When writing code, you will quickly get used to the fact that pretty much any character or word you enter will have a special meaning. For instance, you will see a lot of **keywords** in C#, like class, namespace, public and many more. You will also see that the compiler makes sure that you are using these keywords, as well as your own variables and methods, in the correct way. C# is a fairly strict language and the compiler will help you make sure that everything is entered the way it should be. However, you do have a single possibility to write whatever you like, thanks to the concept of **code comments**.

You may have already experienced comments in some of the code you have seen, be it C# or any other programming language - the concept of comments in code is pretty universal. The way they are written varies a lot though, so let's have a look at the type of comments you can use in your C# code.

## **Single-line comments**

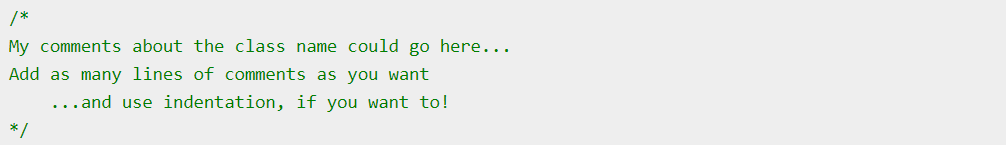
The most basic type of comment in C# is the single-line comment. As the name indicates, it turns a single line into a comment - let's see how it might look:



So that's it: Prefix your lines with two forward slashes (//) and your text goes from something the compiler will check and complain about, to something the compiler completely ignores. And while this only applies to the prefixed line, you are free to do the same on the next line, essentially using single-line comments to make multiple comment lines:

## **Multi-line comments**

In case you want to write multiple lines of comments, it might make more sense to use the multi-line comment variant that C# offers. Instead of having to prefix every line, you just enter a start and stop character sequence - everything in between is treated as comments:



Use the start sequence of forward-slash-asterisk (/\*), write whatever you like, across multiple lines or not, and then end it all with the end sequence of asterisk-forward-slash (\*/). In between these markers, you can write whatever you want.

As with pretty much any other programming related subject, whether to use multiple single-line comments or one multi-line comment is often debated. Personally, I use both, for various situations - in the end, it's all up to you!

## **Documentation comments**

Documentation Comments (sometimes referred to as XML Documentation Comments) looks like regular comments, but with embedded XML. Just like with regular comments, they come in two forms: Single-line and multi-line. You also write them the same way, but with an extra character. So, single-line XML Documentation Comments uses three forward slashes (///) instead of two, and the multi-line variant gets an extra asterisk added in the start delimiter. Let's see how it looks:

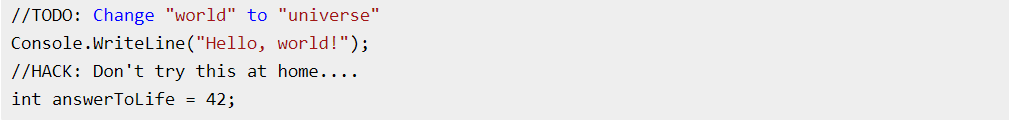
// See Comments Example 1

Here you can see both variants - single-line and multi-line. The result is the same, but the first variant tends to be the most commonly used for documentation comments.

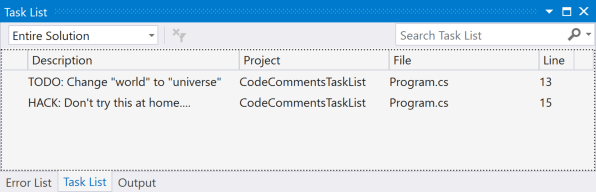
Documenting your types and their members with documentation comments is a pretty big subject, and therefore it will be covered more in depth in a later article, but now you know how they look!

## **Code comments & the Task list**

If you're using Visual Studio, you can actually get some help tracking your code comments. In the Task List window (access it from the menu **View** > **Task List**) your comments will appear if they use the special, but very simple Task List comment syntax:



So if the single-line comment is immediately followed by TODO or HACK, it will appear in the Task List of Visual Studio, like this:



And there are more types - depending on the version of Visual Studio you're using, it will respond to some or all of the following comment tokens:

* TODO
* HACK
* NOTE
* UNDONE

You can even add your own tokens, if you want to - just follow the steps described in [this article](https://docs.microsoft.com/en-us/visualstudio/ide/using-the-task-list).

## **Summary**

Code comments are extremely useful in documenting your code or for the purpose of leaving clues to your self or your potential colleagues on how stuff works. As an added benefit, they're great when you need to test something quickly - just copy a line and comment out the original line and you can see how it works now. If you're not happy with the result, you can just delete the new line and uncomment the original line and you're back to where you started.

And don't worry about the end-user snooping through your comments - they are, as already mentioned, completely ignored by the compiler and therefore not in anyway included in your final DLL or EXE file. Code comments are your personal free-space when programming, so use it in any way you want to.

# The if statement

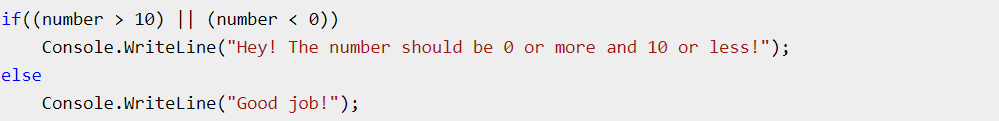
One of the single most important statements in every programming language is the if statement. Being able to set up conditional blocks of code is a fundamental principal of writing software. In C#, the if statement is very simple to use. If you have already used another programming language, chances are that you can use the if statement in C# straight away. In any case, read on to see how it's used. The if statement needs a boolean result, that is, true or false. In some programming languages, several datatypes can be automatically converted into booleans, but in C#, you have to specifically make the result boolean. For instance, you can't use if(number), but you can compare a number to something, to generate a true or false, like we do later on.

In the previous chapter we looked at variables, so we will expand on one of the examples to see how conditional logic can be used.

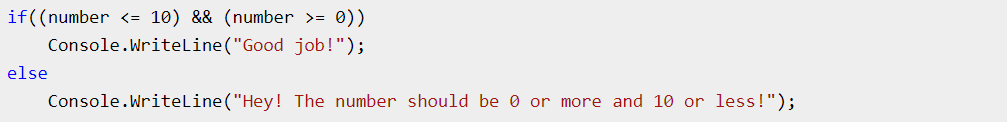
// See Control Structures Example 1

We use 2 if statements to check if the entered number is between 0 and 10, and a companion of the if statement: The else keyword. Its meaning should be obvious to anyone speaking English - it simply offers an alternative to the code being executed if the condition of the if statement is not met.

As you may have noticed, we don't use the { and } characters to define the conditional blocks of code. The rule is that if a block only contains a single line of code, the block characters are not required. Now, this seems like a lot of lines to simply check a number, doesn't it? It can be done with fewer lines of code, like this:



We put each condition in a set of parentheses, and then we use the || operator, which simply means "or", to check if the number is either more than 10 OR less than 0. Another operator you will be using a lot is the AND operator, which is written like this: &&. Could we have used the AND operator instead? Of course, we simply turn it around a bit, like this:



This introduces a couple of new operators, the "less than or equal to" and the "greater than or equal to".

# The switch statement

The switch statement is like a set of if statements. It's a list of possibilities, with an action for each possibility, and an optional default action, in case nothing else evaluates to true. A simple switch statement looks like this:

// See Control Structures Example 2

The identifier to check is put after the switch keyword, and then there's the list of case statements, where we check the identifier against a given value. You will notice that we have a break statement at the end of each case. C# simply requires that we leave the block before it ends. In case you were writing a function, you could use a return statement instead of the break statement.

In this case, we use an integer, but it could be a string too, or any other simple type. Also, you can specify the same action for multiple cases. We will do that in the next example too, where we take a piece of input from the user and use it in our switch statement:

// See Control Structures Example 3

In this example, we ask the user a question, and suggest that they enter either yes, no or maybe. We then read the user input, and create a switch statement for it. To help the user, we convert the input to lowercase before we check it against our lowercase strings, so that there is no difference between lowercase and uppercase letters.

Still, the user might make a typo or try writing something completely different, and in that case, no output will be generated by this specific switch statement. Enter the default keyword!

// See Control Structures Example 4

If none of the case statements has evaluated to true, then the default statement, if any, will be executed. It is optional, as we saw in the previous examples.

Loops

Another essential technique when writing software is looping - the ability to repeat a block of code X times. In C#, they come in 4 different variants, and we will have a look at each one of them.

The while loop

The while loop is probably the most simple one, so we will start with that. The while loop simply executes a block of code as long as the condition you give it is true. A small example, and then some more explanation:

// See Control Structures - Loops - Example 1

Try running the code. You will get a nice listing of numbers, from 0 to 4. The number is first defined as 0, and each time the code in the loop is executed, it's incremented by one. But why does it only get to 4, when the code says 5? For the condition to return true, the number has to be less than 5, which in this case means that the code which outputs the number is not reached once the number is equal to 5. This is because the condition of the while loop is evaluated before it enters the code block.

## **The do loop**

The opposite is true for the do loop, which works like the while loop in other aspects though. The do loop evaluates the condition after the loop has executed, which makes sure that the code block is always executed at least once.

// See Control Structures - Loops - Example 2

The output is the same though - once the number is more than 5, the loop is exited.

## **The for loop**

The for loop is a bit different. It's preferred when you know how many iterations you want, either because you know the exact amount of iterations, or because you have a variable containing the amount. Here is an example of the for loop.

// See Control Structures - Loops - Example 3

This produces the exact same output, but as you can see, the for loop is a bit more compact. It consists of 3 parts - we initialize a variable for counting, set up a conditional statement to test it, and increment the counter (++ means the same as "variable = variable + 1").

The first part, where we define the i variable and set it to 0, is only executed once, before the loop starts. The last 2 parts are executed for each iteration of the loop. Each time, i is compared to our number variable - if i is smaller than number, the loop runs one more time. After that, i is increased by one.

Try running the program, and afterwards, try changing the number variable to something bigger or smaller than 5. You will see the loop respond to the change.

## **The foreach loop**

The last loop we will look at, is the foreach loop. It operates on collections of items, for instance arrays or other built-in list types. In our example we will use one of the simple lists, called an ArrayList. It works much like an array, but don't worry, we will look into it in a later chapter.

// See Control Structures - Loops - Example 4

Okay, so we create an instance of an ArrayList, and then we add some string items to it. We use the foreach loop to run through each item, setting the name variable to the item we have reached each time. That way, we have a named variable to output. As you can see, we declare the name variable to be of the string type – you always need to tell the foreach loop which datatype you are expecting to pull out of the collection. In case you have a list of various types, you may use the object class instead of a specific class, to pull out each item as an object.

When working with collections, you are very likely to be using the foreach loop most of the time, mainly because it’s simpler than any of the other loops for these kind of operations.

# Introduction to C# classes

In lots of programming tutorials, information about classes will be saved for much later. However, since C# is all about Object Oriented programming and thereby classes, we will look at a basic introduction to the most important features now.

First of all, a class is a group of related methods and variables. A class describes these things, and in most cases, you create an instance of this class, now referred to as an object. On this object, you use the defined methods and variables. Of course, you can create as many instances of your class as you want to. Classes, and Object Oriented programming in general, is a huge topic. We will cover some of it in this chapter as well as in later chapters, but not all of it.

In the Hello world chapter, we saw a class used for the first time, since everything in C# is built upon classes. Let's expand our Hello world example with a class we build on our own:

// See Classes - Introduction to C# classes - Example 1

Okay, lots of new stuff here, but almost all of it is based on things we've already used earlier in this tutorial. As you can see, we have defined a new class, called Car. It's declared in the same file as our main application, for an easier overview, however, usually new classes are defined in their own files. It defines a single variable, called color, which of course is used to tell the color of our car. We declared it as private, which is good practice - accessing variables from the outside should be done using a property. The Color property is defined in the end of the class, giving access to the color variable.

Besides that, our Car class defines a constructor. It takes a parameter which allows us to initialize Car objects with a color. Since there is only one constructor, Car objects can only be instantiated with a color. The Describe() method allows us to get a nice message with the single piece of information that we record about our car. It simply returns a string with the information we provide.

Now, in our main application, we declare a variable of the type Car. After that, we create a new instance of it, with "Red" as a parameter. According to the code of our class, this means that the color red will be assigned as the color of the car. To verify this, we call the Describe() method, and to show how easily we can create several instances of the same class, we do it again, but with another color. We have just created our first functional class and used it.

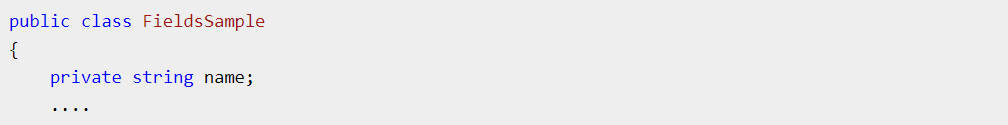
In the following chapters, concepts like: properties, constructors, and visibility will be explained in more depth.

# Fields

One of the most basic building blocks of a class is a field. It's just like a variable, which we talked about previously, but defined on the class level instead of the method level. The difference is quite important and it has everything to do with the concept of scopes, which decides from where a variable can be accessed from: A local variable (defined inside a method) can only be accessed from this specific method, while a class field can be accessed from all methods of a class and even from methods in other classes if the visibility allows it.

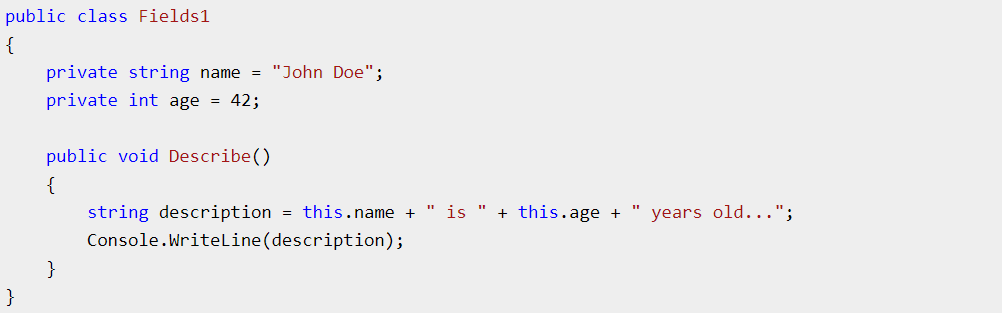
In other words, the difference between a variable and a field is pretty much where it's declared. A class-level variable is referred to as a field, while a method-level variable is usually just referred to as a variable.

Fields are often declared near the top of the class and their visibility are often set to private (we'll discuss visibility later in this chapter). It could look like this:



We now have a class-level variable called "name", which can be accessed from all methods of this class. It can't be accessed from outside the class, but only because we have marked it as **private**. You are free to declare your fields as **protected** if you want to access them from derived classes or even **public** if you want to access them from anywhere, but keep in mind that **the recommended way of accessing fields from outside a class is through properties**, which we'll discuss in the next article.

In our example above, or "name" variable doesn't have an initial value, meaning that you will have to assign something to it before you can use it. If you already know which value your field should start with, you can easily assign it at the same time as declaring it:



In the Describe() method (and don't worry, we'll get to methods in one of the next articles) we declare a local variable called "description", with a value based on our two declared fields. The "description" variable is a great example of a variable which should always be a variable and never a field: It's temporary and only relevant to the method which uses it, where the fields could easily be relevant in other methods of the class.

## **Summary**

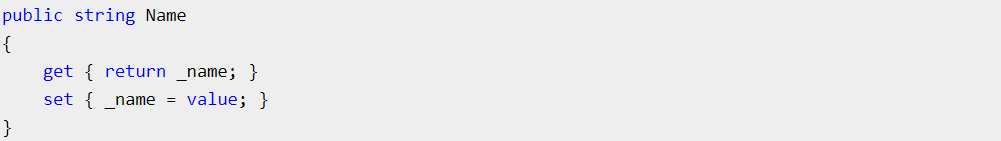
Fields act a bit like global variables because they can be used to store data which can then be accessed from anywhere inside the class. They can be accessed from outside the declaring class as well, but normally, properties are used for this purpose. We'll talk about properties in the next article.

# Properties

In the previous article, we discussed **fields**. They are like global variables for a class, allowing you to access them from all methods. We also briefly discussed the fact that fields CAN be accessed from other classes if they are marked as **public**, but that this is generally not recommended. For variables/fields that you wish to access from outside your class, you should instead use **properties**.

When you declare a field as public, you are giving complete access to it from the outside - other classes can do whatever they want with it, without any notice to the declaring class. Properties gives the control back to the declaring class, by specifying whether a field is read or write-only and even allowing the declaring class to check and manipulate the value before returning or assigning it to the field.

A property looks a bit like a crossover between a field and a method, because it's declared much like a field with visibility, a data type and a name, but it also has a body, like a method, for controlling the behavior:



Notice the special **get** and **set** keywords. They are used exclusively for properties, to control the behavior when reading (get'ing) and writing (set'ing) the field. You can have properties with only a get OR a set implementation, to create read-only or write-only properties, and you can even control the visibility of the get or set implementations, e.g. to create a property which can be read from anywhere (public) but only modified from inside the declaring class (private).

You will also notice that I refer to a field called \_name. You will have to declare that in your class as well, so that your property can use it. A common usage pattern for fields and properties will look like this:

// See Classes - Properties - Example 1

You can now see how the field and the property works together: The **get method** will return the value of the **\_name** field, while the **set method** will assign the passed value to the **\_name** field. In the set method, we use the special keyword value which, in this specific situation, will refer to the value passed to the property.

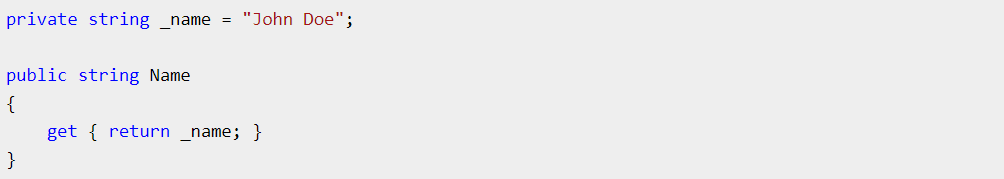
So, this is pretty much as basic as it gets and at this point, we don't do anything that couldn't be accomplished with a simple public field. But at a later point, you may decide that you want to take more control of how other classes can work with the name and since you have implemented this as a property, you are free to modify the implementation without disturbing anyone using your class. For instance, the Name property could be modified to look like this:

// See Classes - Properties - Example 2

The **get method** now enforces that the value returned is always in UPPERCASE, no matter which case the backing field (\_name) is in. In the **set method**, we have added a couple of lines of code to check whether the passed value contains a space, because we have decided that the name should always consist of both a first and a last name - if this is not the case, an exception is thrown. This is all very crude and simplified, but it should illustrate the full level of control you get when using properties.

## **Read-only properties**

Most properties you'll see in the examples of this tutorial will be both readable and writeable, because that's the most common usage of properties, but it doesn't always have to be like that. First of all, you can declare a property with only a get-method, like this:



In this case, you can no longer change the "Name" property - you can only read it and the compiler will throw an error if you try to assign a value to it. You can still change the value of it from inside the class though, since you can simply assign a new value to the backing field "\_name". Doing it that way kind of negates one of the biggest advantage to properties though: The ability to always control whether a value can be accepted. As we already talked about, the set-method is a great way to perform validation of the value, but if you assign a new value to the \_name field from multiple places, because the property is read-only, you don't get this validation.

Fortunately for us, C# offers a solution to this: You can define a set method on the property, but limit its visibility, using e.g. the **private** or the **protected** keyword. This will give you the best of both worlds, where you can still assign a value to the property from inside the class (or any inherited class if you use the protected keyword) and have it validated accordingly. Here's an example:

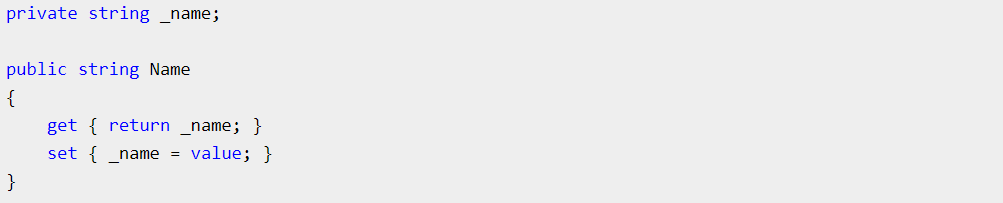
// See Classes - Properties - Example 3

The key difference here is simply the "private" keyword right in front of the "set" keyword, and as mentioned, you can replace it with e.g. protected or internal, depending on your needs.

## **Auto-implemented properties**

In some cases, you don't need all the control over a field and it can feel cumbersome to implement both a field and a property with the get and set methods not doing anything besides what we saw in the first example. You may be tempted to simply declare your variable as a public field to avoid all this extra hassle. **But don't do that!** Fortunately for all of us, Microsoft decided to add auto-implemented properties in C# version 3, which will save you several lines of code. Just consider the difference:

**Regular property with a declared backing field:**



**The exact same behavior, but with an auto-implemented property:**

public string Name { get; set; }

Notice that the get and set methods are empty and that no private backing field is declared - in other words, we can now accomplish the exact same behavior as in the first example but with a single line of code! Bear in mind that the private backing field will still exist at runtime - it will be auto-implemented by the compiler, as the name implies. Should you later decide that you need more control of this specific property, you can simply change it to a regular field/property combination with the desired implementation of the get and set methods.

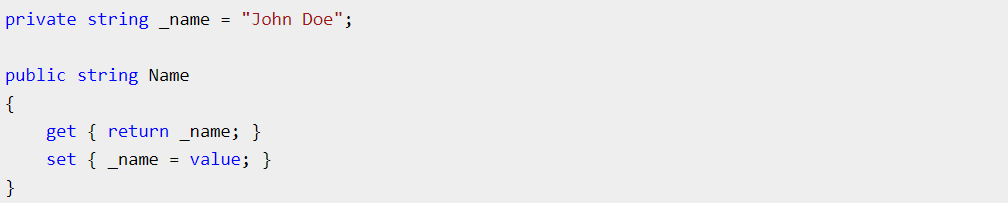
Notice that you are still left with an important mechanism of control from regular properties when using auto-implemented properties: You can leave out the set keyword to create a read-only property, e.g. like this:



Write-only properties are not allowed when using auto-implemented properties.

### **Auto-implemented properties with default values**

Prior to C# version 6, you could not define a default value for an auto-implemented property - for that, you would need a declared backing field, which would allow you to initialize the variable with a value:



But in C# version 6, Microsoft finally added the ability to initialize an auto-implemented property with a default value, like this:

public string Name { get; set; } = "John Doe";

## **Expression-bodied properties**

Another property-related feature Microsoft implemented in C# 6.0 and 7.0 is the expression bodied members. It simply allows you to write single-line expressions for your properties and methods - in this case, let's look at how to use it for your get/set methods in a way that takes up less space and requires slightly less typing:

private string name;

public string Name

{

get => name;

set => name = value;

}

If your property is read-only, the syntax can be even shorter:

public string Name => "John Doe";

Of course this also works if you need to actually do something before returning the value, like this:

public string Name { get; set; } = "John Doe";

public string FirstName => this.Name.Substring(0, this.Name.IndexOf(" "));

As you can see, this allows you to define a get method but without the actual get and return keywords, while encouraging you to keep it all in one line instead of multiple lines.

## **Summary**

Properties gives your classes more control over how fields can be accessed and manipulated, and they should always be used when you want to give access to fields from outside of the declaring class.

Methods (functions)

While properties and fields can be considered passive parts of a class, methods are active. They will perform one or several actions and optionally return a result. In other programming languages they are sometimes referred to as functions or even "funcs", but in C#, where they belong to a class, they are called methods. Methods are very useful because they allow you to encapsulate a piece of functionality in a method which you can then call again from several places.

A method is defined like this:

<visibility> <return type> <name>(<parameters>)

{

<method code>

}

Here's a very basic example:

public int AddNumbers(int number1, int number2)

{

return number1 + number2;

}

This very basic method will add two numbers and return the result. Let's step through the various parts of it:

* **public** is the visibility (more on that later in this tutorial).
* **int** is the return type. If you don't want your method to return anything, use the **void** keyword instead.
* **AddNumbers** is the name of the method.
* (**int number1, int number2**) - these are the parameters (more on those later). Parameters are optional, so you are free to leave the space between the parentheses empty. The parentheses are not optional though.
* Inside the method (between the curly braces), you will find the actual code of the method. It can be one or many lines of code.

To call a method, simply write it's name followed by a set of parentheses. Inside the parentheses, you should write the parameters (if the method accepts any), like this:

AddNumbers(3, 39);

Since methods are defined on classes, you may want to call a method on another class than the one you are currently in. If so, you should prefix the method call with the name of the object, or in case of a static method (more on those later), the name of the class. Here's an example where we call the AddNumbers() method, which has been placed in another class called MathHelper:

public void DoMath()

{

MathHelper mathHelper = new MathHelper(); // mathHelper is the method call

int result = mathHelper.AddNumbers(4, 38);

Console.WriteLine(result);

}

## **Method return types**

Let's talk more about return types. In the examples above, we defined a method with an integer as the return type, but you are free to return any other kind of C# data type. In fact, you can even declare a method which doesn't return anything, as we saw with our **DoMath()** method above. Notice that I have substituted **int** with the **void** keyword, which means that this method is not supposed to return anything. In some programming languages, functions without a return type are referred to as **procedures**, but in C#, they are always called methods.

You should be aware that when you declare a return type for a method, you HAVE to return something - otherwise, the compiler will immediately complain:

public int AddNumbers(int number1, int number2)

{

Console.WriteLine(number1 + number2);

}

Compiler error: AddNumbers(int, int)': not all code paths return a value

This means that you need one (or several) return keywords inside your method if it has a declared return type. You may need more than one for situations where you have multiple possible code paths, like this:

public int AddLargeNumbers(int number1, int number2)

{

if((number1 > 1000) && (number2 > 1000))

{

return number1 + number2;

}

return 0;

}

In this case, we need the second return statement as well - if it's omitted, the compiler will complain, because our conditional statement could prevent the first return statement from being hit.

## **Summary**

Methods allow you to encapsulate and reuse functionality from several places. By supplying different parameters to a method, you can get different results. We have used some parameters in the examples above, but in the next article, we'll dig much deeper into the subject of method parameters.

# Method parameters

In the previous article, we talked about methods and we got a brief introduction to the concept of method/function parameters. In this article, we'll dig deeper into this subject in all its variations. As mentioned, methods can work without parameters, but usually they will have one or more parameters, which will help the method in accomplishing its task.

We already saw a very simple usage scenario for parameters in the previous article: Our AddNumbers() method, which takes two numbers as parameters and returns the sum of these two numbers:

public int AddNumbers(int number1, int number2)

{

return number1 + number2;

}

This goes a long way in showing what a clever concept methods is, because with methods, you can encapsulate the functionality in the method, but be able to affect the result when calling this method through the parameters:

AddNumbers(2, 3);

Result: 5

AddNumbers(38, 4);

Result: 42

This is the basic type of parameters, but let's talk more about all the various modifiers and options you can use to change the behavior of the parameters.

Please notice that in this article, we'll really dig into all the various types of parameters and how they can help you, but if you're just getting started with C# and you just want to see some results, the following might be a bit too complex and technical for now. Feel free to skip the rest of the article and return later when you're ready.

## **Optional parameters**

By default, when calling a method with one or several parameters, you are forced to supply values for all of these parameters. However, in some situations, you may need to make one or several parameters optional. In some programming languages, you would be allowed to do that simply by marking the parameter as optional, but in C#, a parameter is made optional by supplying a default value for it in the method declaration. This is actually a nice solution, because it saves you from writing extra code to deal with situations where the parameter is not supplied by the caller.

Here's an example of a method with an optional parameter:

public int AddNumbers(int number1, int number2, int number3 = 0)

{

return number1 + number2 + number3;

}

The last parameter (number3) is now optional, because we have provided a default value for it (0). When calling it, you can now supply two or three values, like this:

public void Main(string[] args)

{

AddNumbers(38, 4);

AddNumbers(36, 4, 2);

}

You can make more than one parameter optional - in fact, your method may consist of only optional parameters if needed. Just remember that **optional parameters must come last in the method declaration** - not first or in between non-optional parameters.

### **The params modifier**

As an alternative to a number of optional parameters, you can use the params modifier to allow an arbitrary number of parameters. It could look like this:

public void GreetPersons(params string[] names) { }

Calling it could then look like this:

GreetPersons("John", "Jane", "Tarzan");

Another advantage of using the params approach, is that you are allowed to pass zero parameters to the method as well. Methods with params can also take regular parameters, as long as the parameter with the params modifier is the last one. Besides that, only one parameter using the params keyword can be used per method. Here is complete example where we use the params modifier to print a variable number of names with our **GreetPersons()** method:

public void Main(string[] args)

{

GreetPersons("John", "Jane", "Tarzan");

}

public void GreetPersons(params string[] names)

{

foreach(string name in names)

Console.WriteLine("Hello " + name);

}

## **Parameter types: value or reference**

C#, and other programming languages as well, differ between two parameter types: **"by value"** and **"by reference"**. The default in C# is "by value", which basically means that when you pass a variable to a method call, you are actually sending a copy of the object, instead of a reference to it. This also means that you can make changes to the parameter from inside the method, without affecting the original object you passed as a parameter.

We can easily demonstrate this behavior with an example:

public void Main(string[] args)

{

int number = 20;

AddFive(number);

Console.WriteLine(number);

}

public void AddFive(int number)

{

number = number + 5;

}

We have a very basic method called AddFive() which will add 5 to the number we pass to it. So in our Main() method, we create a variable called number with the value of 20 and then call the AddFive() method. Now on the next line, when we write out the number variable, one might expect that value is now 25, but instead, it remains 20. Why? Because by default, the parameter was passed in as a copy of the original object (by value), so when our AddFive() method worked on the parameter, it was working on a copy and thereby never affecting the original variable.

There are currently two ways of changing this behavior, so that our AddFive() method is allowed to modify the original value: We can use the **ref modifier** or the **in/out modifiers**.

### **The ref modifier**

The ref modifier is short for "reference" and it basically just changes the behavior of the parameter from the default behavior of "by value" to "by reference", meaning that we are now passing in a reference to the original variable instead of a copy of its value. Here's a modified example:

public void Main(string[] args)

{

int number = 20;

AddFive(ref number);

Console.WriteLine(number);

}

public void AddFive(ref int number)

{

number = number + 5;

}

You will notice that I have added the "ref" keyword in two places: In the method declaration and when passing in the parameter in the method call. With this change, we now get the behavior we originally might have expected - the result is now 25 instead of 20, because the ref modifier allows the method to work on the actual value passed in as a parameter instead of a copy.

### **The out modifier**

Just like the **ref modifier**, the **out modifier** ensures that the parameter is passed by reference instead of by value, but there's a major difference: When using the ref modifier, you pass in an initialized value which you may choose to modify inside the method, or leave it as it is. On the other hand, when using the **out modifier**, you are forced to initialize the parameter inside the method. This also means that you can pass in uninitialized values when using the out modifier - the compiler will complain if you are trying to leave a method with an out parameter without assigning a value to it.

In C#, a method can only return one value, but if you use the out modifier, you are able to circumvent this by passing in several parameters with the out modifier - when the method has been called, all out parameters will have been assigned. Here's an example, where we pass in two numbers and then, using out modifiers, return both an addition and a subtraction using these numbers:

public void Main(string[] args)

{

int addedValue, subtractedValue;

DoMath(10, 5, out addedValue, out subtractedValue);

Console.WriteLine(addedValue);

Console.WriteLine(subtractedValue);

}

public void DoMath(int number1, int number2, out int addedValue, out int subtractedValue)

{

addedValue = number1 + number2;

subtractedValue = number1 - number2;

}

Output:

15

5

As you can see in the beginning of the example, I declare the two variables (addedValue and subtractedValue) before passing them as out parameters. This was a requirement in previous versions of the C# language, but from C# version 6, you can declare them directly in the method call, like this:

DoMath(10, 5, out int addedValue, out int subtractedValue);

Console.WriteLine(addedValue);

Console.WriteLine(subtractedValue);

### **The in modifier**

Just like the **out modifier**, the **in modifier** ensures that the parameter is passed as a reference instead of a copy of the value, but unlike the out modifier, the **in** modifier will prevent you from making any changes to the variable inside the method.

Your first thought might be: If I can't change the value of the parameter, then I might as well just pass it in as a regular parameter, where changes won't affect the original variable. And you're right, the result would appear to be exactly the same, but there's still a very valid reason for using the in modifier: By passing it as a reference instead of a value, you are saving resources because the framework doesn't have to spend time creating a copy of the object when passing it to the method like a regular parameter.

In most cases, this won't make much of a difference, because most parameters are simple strings or integers, but in situations where you repeatedly call the same method many times in a loop or where you pass in parameters with large values, e.g. very large strings or structs, this may give you a nice performance boost.

Here's an example where we use the in modifier:

public void Main(string[] args)

{

string aVeryLargeString = "Lots of text...";

InMethod(aVeryLargeString);

}

public void InMethod(in string largeString)

{

Console.WriteLine(largeString);

}

In our method, InMethod(), the largeString parameter is now a read-only reference to the original variable (aVeryLargeString), so we're able to use it, but not modify it. If we try, the compiler will complain:

public void InMethod(in string largeString)

{

largeString = "We can't do this...";

}

Error: Cannot assign to variable 'in string' because it is a readonly variable

## **Named parameters**

As you have seen in all of the examples above, each parameter has a unique name in the method declaration. This allows you to reference the parameter inside the method. However, when calling the method, you don't use these names - you just supply the values in the same order as they are declared. This is not a problem for simple methods which takes 2-3 parameters, but some methods are more complex and requires more parameters. In those situations, it can be quite hard to figure out what the various values in a method call refers to, like in this example:

PrintUserDetails(1, "John", 42, null);

It's not very clear what the various parameters means in this method call, but if you look at the method declaration, you can see it:

public void PrintUserDetails(int userId, string name, int age = -1, List<string> addressLines = null)

{

// Print details...

}

But it's annoying if you constantly have to look up the method declaration to understand what the parameters are doing, so for more complex methods, you can supply the parameter names directly in the method call. This also allows you to supply the parameter names in any order, instead of being forced to use the declaration order. Here's an example:

PrintUserDetails(name: "John Doe", userId: 1);

As an added bonus, this will allow you to supply a value for any of your optional parameters, without having to supply values for all previous optional parameters in the declaration. In other words, if you want to supply a value for the **addressLines** parameter in this example you would also have to provide a value for the **age** parameter, because it comes first in the method declaration. However, if you use named parameters, order no longer matters and you can just supply values for the required parameters as well as one or several of the optional parameters, e.g. like this:

PrintUserDetails(addressLines: new List<string>() { }, name: "Jane Doe", userId: 2);

Here's the complete example of using named parameters:

public void Main(string[] args)

{

PrintUserDetails(1, "John", 42, null);

PrintUserDetails(name: "John Doe", userId: 1);

PrintUserDetails(addressLines: new List<string>() { }, name: "Jane Doe", userId: 2);

}

public void PrintUserDetails(int userId, string name, int age = -1, List<string> addressLines = null)

{

// Print details...

Console.WriteLine(name + " is " + age + " years old...");

}

## **Summary**

As you can see from this article, parameters come in many forms and types. Fortunately, you'll come along way with plain, old regular parameters, but once you start to dig deeper into the C# language, you will benefit from knowing all the types and modifiers as explained in this article

# Constructors and destructors

Constructors are special methods, used when instantiating a class. A constructor can never return anything, which is why you don't have to define a return type for it. A normal method is defined like this:

public string Describe()

A constructor can be defined like this:

public Car()

In our example for this chapter, we have a Car class, with a constructor which takes a string as argument. Of course, a constructor can be overloaded as well, meaning we can have several constructors, with the same name, but different parameters. Here is an example:

public Car()

{

}

public Car(string color)

{

this.color = color;

}

A constructor can call another constructor, which can come in handy in several situations. Here is an example:

public Car()

{

Console.WriteLine("Constructor with no parameters called!");

}

public Car(string color) : this()

{

this.color = color;

Console.WriteLine("Constructor with color parameter called!");

}

If you run this code, you will see that the constructor with no parameters is called first. This can be used for instantiating various objects for the class in the default constructor, which can then be called from other constructors from the class. If the constructor you wish to call takes parameters, you can do that as well. Here is a simple example:

public Car(string color) : this()

{

this.color = color;

Console.WriteLine("Constructor with color parameter called!");

}

public Car(string param1, string param2) : this(param1)

{

}

If you call the constructor which takes 2 parameters, the first parameter will be used to invoke the constructor that takes 1 parameter.

## **Destructors**

Since C# is garbage collected, meaning that the framework will free the objects that you no longer use, there may be times where you need to do some manual cleanup. A destructor, a method called once an object is disposed, can be used to cleanup resources used by the object. Destructors doesn't look very much like other methods in C#. Here is an example of a destructor for our Car class:

~Car()

{

Console.WriteLine("Out..");

}

Once the object is collected by the garbage collector, this method is called.

# Method overloading

A lot of programming languages support a technique called default/optional parameters. It allows the programmer to make one or several parameters optional, by giving them a default value. It's especially practical when adding functionality to existing code.

For instance, you may wish to add functionality to an existing function, which requires one or more parameters to be added. By doing so, you would break existing code calling this function, since they would now not be passing the required amount of parameters. To work around this, you could define the newly added parameters as optional, and give them a default value that corresponds to how the code would work before adding the parameters.

Default parameters were introduced in C# version 4.0, but up until that, C# coders have been using a different technique, which basically does the same, called method overloading. It allows the programmer to define several methods with the same name, as long as they take a different set of parameters. When you use the classes of the .NET framework, you will soon realize that method overloading is used all over the place. A good example of this, is the Substring() method of the String class. It has an extra overload, like this:

string Substring (int startIndex)

string Substring (int startIndex, int length)

You can call it with either one or two parameters. If you only call it with one parameter, the length parameter is assumed to be the rest of the string, saving us time whenever we simply want to get the last part of a string.

So, by defining several versions of the same function, how do we avoid having the same code several places? It's actually quite simple: We let the simple versions of the method make the complex version of it do all the work. Consider the following example:

class SillyMath

{

public static int Plus(int number1, int number2)

{

return Plus(number1, number2, 0); //El mas corto llama al mas largo

}

public static int Plus(int number1, int number2, int number3)

{

return number1 + number2 + number3;

}

}

We define a Plus method, in two different versions. The first one takes two paramaters, for adding two numbers, while the second version takes three numbers. The actual work is done in the version that takes three numbers - if we only wish to add two, we call the three parameter version, and simply use 0 as the third paramater, acting as a default value. I know, I know, it's a silly example, as indicated by the name of the class, but it should give you an idea about how it all works.

Now, whenever you feel like doing advanced math by adding a total of four numbers (just kidding here), it's very simple to add a new overload:

class SillyMath

{

public static int Plus(int number1, int number2)

{

return Plus(number1, number2, 0);

}

public static int Plus(int number1, int number2, int number3)

{

return Plus(number1, number2, number3, 0);

}

public static int Plus(int number1, int number2, int number3, int number4)

{

return number1 + number2 + number3 + number4;

}

}

The cool thing about this, is that all your existing calls to the Plus method will continue working, as if nothing had been changed. The more you use C#, the more you will learn to appreciate method overloading.

# Visibility

The visibility of a class, a method, a variable or a property tells us how this item can be accessed. The most common types of visibility are private and public, but there are actually several other types of visibility within C#. Here is a complete list, and although some of them might not feel that relevant to you right now, you can always come back to this page and read up on them:

**public** - the member can be reached from anywhere. This is the least restrictive visibility. Enums and interfaces are, by default, publicly visible.

**protected** - members can only be reached from within the same class, or from a class which inherits from this class.

**internal** - members can be reached from within the same project only.

**protected internal** - the same as internal, except that classes which inherit from this class can reach its members; even from another project.

**private** - can only be reached by members from the same class. This is the most restrictive visibility. Classes and structs are by default set to private visibility.

So for instance, if you have two classes: Class1 and Class2, private members from Class1 can only be used within Class1. You can't create a new instance of Class1 inside of Class2, and then expect to be able to use its private members.

If Class2 inherits from Class1, then only non-private members can be reached from inside of Class2.

# Static members

As we saw in a previous chapter, the usual way to communicate with a class is to create a new instance of the class and then work on the resulting object. In most cases, this is what classes are all about; the ability to instantiate multiple copies of the same class and then use them differently in some way. However, in some cases, you might like to have a class which you may use without instantiating it, or at least a class where you can use members of it without creating an object for it. For instance, you may have a class with a variable that always remains the same no matter where and how it's used. This is called a static member, because it remains the same.

A class can be static, and it can have static members, both functions and fields. A static class can't be instantiated, so in other words, it will work more as a grouping of related members than an actual class. You may choose to create a non-static class instead, but let it have certain static members. A non-static class can still be instantiated and used like a regular class, but you can't use a static member on an object of the class. A static class may only contain static members.

First, here is an example of a static class:

public static class Rectangle

{

public static int CalculateArea(int width, int height)

{

return width \* height;

}

}

As you can see, we use the static keyword to mark the class as static, and then we use it again to mark the method, CalculateArea, as static as well. If we didn't do that, the compiler would complain, since we can't have a non-static member of a static class.

To use this method, we call it directly on the class, like this:

Console.WriteLine("The area is: " + Rectangle.CalculateArea(5, 4));

We could add other helpful methods to the Rectangle class, but perhaps you are wondering why we are passing on width and height to the actual method, instead of storing it inside the class and then pulling them from there when needed? Because it's static! We could store them, but only one set of dimensions, because there is only one version of a static class. This is very important to understand.

Instead, we can make the class non-static, and then have the CalculateArea as a utility function on this class:

public class Rectangle

{

private int width, height;

public Rectangle(int width, int height)

{

this.width = width;

this.height = height;

}

public void OutputArea()

{

Console.WriteLine("Area output: " + Rectangle.CalculateArea(this.width, this.height));

}

public static int CalculateArea(int width, int height)

{

return width \* height;

}

}

As you can see, we have made the class non-static. We have also added a constructor, which takes a width and a height and assigns it to the instance. Then we have added an OutputArea method, which uses the static method to calculate the area. This is a fine example of mixing static members with non-static members, in a non-static class.

A common usage of static classes, although frowned upon by some people, are utility/helper classes, where you collect a bunch of useful methods, which might not belong together, but don't really seem to fit elsewhere either.

# Inheritance

One of the absolute key aspects of Object Oriented Programming (OOP), which is the concept that C# is built upon, is inheritance, the ability to create classes which inherits certain aspects from parent classes. The entire .NET framework is built on this concept, with the "everything is an object" as a result of it. Even a simple number is an instance of a class, which inherits from the System.Object class, although .NET helps you out a bit, so you can assign a number directly, instead of having to create a new instance of e.g. the integer class.

This subject can be a bit difficult to comprehend, but sometimes it help with some examples, so let's start with a simple one of those:

public class Animal

{

public void Greet()

{

Console.WriteLine("Hello, I'm some sort of animal!");

}

}

public class Dog : Animal

{

}

First, we define an Animal class, with a simple method to output a greeting. Then we define a Dog class, and with a colon, we tell C# that the Dog class should inherit from the Animal class. The beautiful thing about this is that it makes sense in the real world as well - a Dog is, obviously, an Animal. Let's try using the classes:

Animal animal = new Animal();

animal.Greet();

Dog dog = new Dog();

dog.Greet();

If you run this example, you will notice that even though we have not defined a Greet() method for the Dog class, it still knows how to greet us, because it inherits this method from the Animal class. However, this greeting is a bit anonymous, so let's customize it when we know which animal it is:

public class Animal

{

public virtual void Greet()

{

Console.WriteLine("Hello, I'm some sort of animal!");

}

}

public class Dog : Animal

{

public override void Greet()

{

Console.WriteLine("Hello, I'm a dog!");

}

}

Besides the added method on the Dog class, you should notice two things: I have added the virtual keyword to the method on the Animal class, and on the Dog class, I use the override keyword.

In C#, you are not allowed to override a member of a class unless it's marked as virtual. If you want to, you can still access the inherited method, even when you override it, using the base keyword.

public override void Greet()

{

base.Greet();

Console.WriteLine("Yes I am - a dog!");

}

Methods are not the only thing to get inherited, though. In fact, pretty much all class members will be inherited, including fields and properties. Just remember the rules of visibility, as discussed in a previous chapter.

Inheritance is not only from one class to another - you can have a whole hierarchy of classes, which inherits from each other. For instance, we could create a Puppy class, which inherits from our Dog class, which in turn inherits from the Animal class. What you can't do in C#, is to let one class inherit from several other classes at the same time. Multiple inheritance, as it's called, is not supported by C#.

# Abstract classes

Abstract classes, marked by the keyword abstract in the class definition, are typically used to define a base class in the hierarchy. What's special about them, is that you can't create an instance of them - if you try, you will get a compile error. Instead, you have to subclass them, as taught in the chapter on inheritance, and create an instance of your subclass. So when do you need an abstract class? It really depends on what you do.

To be honest, you can go a long way without needing an abstract class, but they are great for specific things, like frameworks, which is why you will find quite a bit of abstract classes within the .NET framework itself. A good rule of thumb is that the name actually makes really good sense - abstract classes are very often, if not always, used to describe something abstract, something that is more of a concept than a real thing.

In this example, we will create a base class for four legged animals and then create a Dog class, which inherits from it, like this:

namespace AbstractClasses

{

class Program

{

static void Main(string[] args)

{

Dog dog = new Dog();

Console.WriteLine(dog.Describe());

Console.ReadKey();

}

}

abstract class FourLeggedAnimal

{

public virtual string Describe()

{

return "Not much is known about this four legged animal!";

}

}

class Dog : FourLeggedAnimal

{

}

}

If you compare it with the examples in the chapter about inheritance, you won't see a big difference. In fact, the abstract keyword in front of the FourLeggedAnimal definition is the biggest difference. As you can see, we create a new instance of the Dog class and then call the inherited Describe() method from the FourLeggedAnimal class. Now try creating an instance of the FourLeggedAnimal class instead:

FourLeggedAnimal someAnimal = new FourLeggedAnimal();

You will get this fine compiler error:

*Cannot create an instance of the abstract class or interface 'AbstractClasses.FourLeggedAnimal'*

Now, as you can see, we just inherited the Describe() method, but it isn't very useful in it's current form, for our Dog class. Let's override it:

class Dog : FourLeggedAnimal

{

public override string Describe()

{

return "This four legged animal is a Dog!";

}

}

In this case, we do a complete override, but in some cases, you might want to use the behavior from the base class in addition to new functionality. This can be done by using the base keyword, which refers to the class we inherit from:

abstract class FourLeggedAnimal

{

public virtual string Describe()

{

return "This animal has four legs.";

}

}

class Dog : FourLeggedAnimal

{

public override string Describe()

{

string result = base.Describe();

result += " In fact, it's a dog!";

return result;

}

}

Now obviously, you can create other subclasses of the FourLeggedAnimal class - perhaps a cat or a lion? In the next chapter, we will do a more advanced example and introduce abstract methods as well. Read on.

# More abstract classes

In the previous chapter, we had a look at abstract classes. In this chapter, we will expand the examples a bit, and throw in some abstract methods as well. Abstract methods are only allowed within abstract classes. Their definition will look like a regular method, but they have no code inside them:

abstract class FourLeggedAnimal

{

public abstract string Describe();

}

So, why would you want to define an empty method that does nothing? Because an abstract method is an obligation to implement that very method in all subclasses. In fact, it's checked at compile time, to ensure that your subclasses has this method defined. Once again, this is a great way to create a base class for something, while still maintaining a certain amount of control of what the subclasses should be able to do. With this in mind, you can always treat a subclass as its baseclass, whenever you need to use methods defined as abstract methods on the baseclass. For instance, consider the following example:

We are calling an previously existing object

namespace AbstractClasses

{

class Program

{

static void Main(string[] args)

{

System.Collections.ArrayList animalList = new System.Collections.ArrayList();

animalList.Add(new Dog());

animalList.Add(new Cat());

foreach(FourLeggedAnimal animal in animalList)

Console.WriteLine(animal.Describe());

Console.ReadKey();

}

}

abstract class FourLeggedAnimal

{

public abstract string Describe(); //you don’t write ‘virtual’

}

class Dog : FourLeggedAnimal

{

public override string Describe()

{

return "I'm a dog!";

}

}

class Cat : FourLeggedAnimal

{

public override string Describe()

{

return "I'm a cat!";

}

}

}

As you can see, we create an ArrayList to contain our animals. We then instantiate a new dog and a new cat and add them to the list. They are instantiated as a Dog and a Cat respectively, but they are also of the type FourLeggedAnimal, and since the compiler knows that subclasses of that class contains the Describe() method, you are actually allowed to call that method, without knowing the exact type of animal. So by typecasting to the FourLeggedAnimal, which is what we do in the foreach loop, we get access to members of the subclasses. This can be very useful in lots of scenarios.

# Interfaces

In previous chapters, we had a look at abstract classes. Interfaces are much like abstract classes and they share the fact that no instances of them can be created. However, interfaces are even more conceptual than abstract classes, since no method bodies are allowed at all. So an interface is kind of like an abstract class with nothing but abstract methods, and since there are no methods with actual code, there is no need for any fields. Properties are allowed though, as well as indexers and events. You can consider an interface as a contract - a class that implements it is required to implement all of the methods and properties. However, the most important difference is that while C# doesn't allow multiple inheritance, where classes inherit more than a single base class, it does in fact allow for implementation of multiple interfaces!

So, how does all of this look in code? Here's a pretty complete example. Have a look, perhaps try it out on your own, and then read on for the full explanation:

// Se coded Example

Let's start in the middle, where we declare the interface. As you can see, the only difference from a class declaration, is the keyword used - interface instead of class. Also, the name of the interface is prefixed with an I for Interface - this is simply a coding standard, and not a requirement. You can call your interfaces whatever you want, but since they are used like classes so much that you might have a hard time telling the difference in some parts of your code, the I prefix makes pretty good sense.

Then we declare the Describe method, and afterwards, the Name property, which has both a get and a set keyword, making this a read and writeable property. You will also notice the lack of access modifiers (public, private, protected etc.), and that's because they are not allowed in an interface - they are all public by default.

Next up is our Dog class. Notice how it looks just like inheriting from another class, with the colon between the class name and the class/interface being subclassed/implemented. However, in this case, two interfaces are implemented for the same class, simply separated by a comma. You can implement as many interfaces as you want to, but in this case we only implement two - our own IAnimal interface, and the .NET IComparable interface, which is a shared interface for classes that can be sorted. Now as you can see, we have implemented both the method and the property from the IAnimal interface, as well as a CompareTo method from the IComparable interface.

Now you might be thinking: If we have to do all the work our self, by implementing the entire methods and properties, why even bother? And a very good example of why it's worth your time, is given in the top of our example. Here, we add a bunch of Dog objects to a list, and then we sort the list. And how does the list know how to sort dogs? Because our Dog class has a CompareTo method that can tell how to compare two dogs. And how does the list know that our Dog object can do just that, and which method to call to get the dogs compared? Because we told it so, by implementing an interface that promises a CompareTo method! This is the real beauty of interfaces.

# Namespaces

In one of the first articles, we briefly discussed namespaces. You probably recognize the keyword, because it's found in most files containing C# code, usually almost in the top. A namespace is essentially a way to group a set of types, e.g. classes, in a named space of its own. When Visual Studio generates a new project for you, it also generates a default namespace in which it places your first file (at least this is true for the Console App project type). It could look like this:

using System;    
  
namespace MyProject    
{    
    class Program    
    {    
 static void Main(string[] args)    
 {    
 // More code below this....

In this case, the namespace "MyProject" is now a part of the application and when using its classes outside of it, you will have to prefix the class name with the namespace name. You see the exact same thing when you want to use something burried deep down in the .NET framework, like this:

System.IO.File.ReadAllText("test.txt");

In this case, we use the ReadAllText() method found on the File class which exists in the System.IO ***namespace***. Of course, it would be tedious to write such a long name each time you wanted to use a class from a namespace, so C# allows you to "import" an entire namespace into the scope of your file with a using statement. Again, you might already know them, because you can usually find them at the top of your C# files. For the example above, if we would need the File class more than once, it would make sense to import the System.IO namespace with a using statement like this:

using System;  
using System.IO;  
// More using statements here...

## **Why do you need namespaces?**

If you have just started programming, you might wonder what we need namespaces for. Why not just put all your classes in the same namespaces so that they are always accessible? You have a valid point, but only if your project is very small. As soon as you start adding more and more classes, it makes very good sense to separate them into namespaces. It simply makes it easier for you to find your code, especially if you places your files in corresponding folders - in fact, if you add a folder to your project and then add a class to it, Visual Studio will automatically put it in a corresponding namespace. So, if you create a folder in MyProject called MyFolder, classes added to this folder will, by default, be placed in a namespace called MyProject.MyFolder.

A great example of why namespaces are needed is the .NET framework itself. Just think if ALL the classes in the framework were just floating around in a global namespace - it would be a mess! Instead, they have organized them nicely, with System as the root namespace for most classes and then sub-namespaces like **System.IO** for input/output stuff, **System.Net** for network related stuff and **System.Net.Mail** for mail-related stuff.

## **Name Conflicts with Namespaces**

As mentioned, namespaces are also there to encapsulate your types (usually classes), so that they can exist within their own domain. This also means that you are free to create classes with the same name as the ones found elsewhere in your project or even in the .NET framework. For instance, you might decide that you need a File class of your own. As we saw in the previous examples, such a class already exists in the System.IO namespace, but you are free to create one in your own namespace, like this:

using System;    
  
namespace MyProject.IO    
{    
    class File    
    {    
 public static void HelloWorld()    
 {    
     Console.WriteLine("Hello, world!");    
 }    
    }    
}

Now when you want to use it in your project, e.g. in your Program.cs Main method (if you're working on a Console App, like I am), you can either write the full name:

MyProject.IO.File.HelloWorld();

But you can also import the namespace, like you can with any other namespace (built-in or user-defined), thanks to the using statement. Here's a more complete example:

using System;  
using MyProject.IO;  
  
namespace MyProject  
{  
    class Program  
    {  
 static void Main(string[] args)  
 {  
     File.HelloWorld();  
 }  
    }  
}

So far, so good! However, what if you also want to use the File class from the System.IO namespace? Well, this is where the trouble starts, because if you import that namespace as well, with a using statement, the compiler no longer knows which File class you're referring to - our own or the one from the System.IO namespace. This can be solved by only importing one of the namespaces (ideally the one from which you use the most types) and then fully qualifying the name of the other one, like in this example:

using System;  
using System.IO;  
  
namespace MyProject  
{  
    class Program  
    {  
 static void Main(string[] args)  
 {  
     MyProject.IO.File.HelloWorld();  
 }  
    }  
}

But it's a bit cumbersome to type each time, especially if your class is even deeper nested in namespaces, e.g. MyProject.FileStuff.IO.File. Fortunately, C# has a solution for that.

## **Using Alias Directive**

To shorten the name of the namespace a lot, you can import the namespace under a different name, with a Using Alias Directive. Notice how I do just that in the next example:

using System;  
using System.IO;  
using MyIO = MyProject.IO;  
  
namespace MyProject  
{  
    class Program  
    {  
 static void Main(string[] args)  
 {  
     File.ReadAllText("test.txt");  
     MyIO.File.HelloWorld();  
 }  
    }  
}

The magic happens in the third line, where I pull in the MyProject.IO namespace and give it a shorter name (MyIO), which can then be used when we want to access types from it. At this point, we're not saving a lot of keystrokes, but again you should imagine even longer names and levels of namespaces, and believe me, they can get quite long and nested.

## **Summary**

Namespaces gives you the opportunity to encapsulate your types into "named spaces", which allows you to get a better structure in your code, as well as have multiple classes with the same name, as long as they exist in separate namespaces.

# Constants (the const keyword)

So far, we have dealt a lot with variables and as the name implies, variables can always be changed. The opposite of that is a constant, introduced in C# with the keyword const. When declaring a constant, you have to immediately assign a value to it and after that, NO changes can be made to the value of this constant. This is great when you have a value which doesn't ever change, and you want to make sure that it's not manipulated by your code, even by accident.

You will find many constants in the framework itself, e.g. in the Math class, where a constant for PI has been defined:

Console.WriteLine(Math.PI);

But of course, the interesting part is to declare some constants of our own. A constant can be defined in the scope of a method, like this:

static void Main(string[] args)  
{  
    const int TheAnswerToLife = 42;  
    Console.WriteLine("The answer to life, the universe and everything: " +  TheAnswerToLife);  
}

However, most constants are declared on the class level, so that they can be accessed (but not changed, of course) from all methods of the class and even outside of the class, depending on the visibility. A constant will act like a static member of the class, meaning that you can access it without instantiating the class. With that in mind, let's try a full example where two constants are defined - a private and a public constant:

using System;  
  
namespace Constants  
{  
    class Program  
    {  
 static void Main(string[] args)  
 {  
     Console.WriteLine("The fake answer to life: " + SomeClass.TheFakeAnswerToLife);  
     Console.WriteLine("The real answer to life: " +  SomeClass.GetAnswer());  
 }  
    }  
  
    class SomeClass  
    {  
 private const int TheAnswerToLife = 42; //only accessible via a public method  
 public const int TheFakeAnswerToLife = 43;  
  
 public static int GetAnswer()  
 {  
     return TheAnswerToLife;  
 }  
    }  
}

Notice how I define a class (SomeClass) with two constants. The first is private, so it can only be access from the class it self, but the other one is public. So, in our main program code, I access both constants differently – first directly, since the fake answer is publicly available, and secondly with the help of the GetAnswer() method.

## **Which types can be used as a constant?**

Since constants has to be declared immediately and can't be changed later on, the value you assign to a constant has to be a constant expression and the compiler must be able to evaluate the value already at compile time. This means that numbers, boolean values and strings can be used just fine for a constant, while e.g. a DateTime object can't be used as a constant.

Since the compiler needs to know the value immediately, it also means that there are some limitations to what you can do when setting the value. For instance, these are perfect examples of what you CAN do:

const int a = 10;    
const float b = a \* 2.5f;  
  
const string s1 = "Hello, world!";    
const string s2 = s1 + " How are you?";

On the other hand, you can't use the result of a method call or a non-constant class member, since these are not constant expressions. Here comes a couple of examples of what you CAN'T do:

// NOT possible:  
const int a = Math.Cos(4) \* 2;  
// Possible:  
const string s1 = "Hello, world!";  
// NOT possible:  
const string s2 = s1.Substring(0, 6) + " Universe";

The difference lies in what the compiler can be expected to know when it reaches your code, e.g. numbers, strings and other constants, in contrast to what it has to execute to get the value for.

### **A constant alternative: The readonly field**

If you're looking for a slightly less restrictive version of a class constant, you may want to have a look at the readonly keyword. It's not available on the method level, but it can be used on the class level, to define a field which can only be modified during declaration or the execution of the constructor method of the class. So, as soon as the object is available for use, the readonly field will have the same value forever and can't be modified by the user. Let's try it out:

class SomeClass  
{  
    private readonly DateTime rightNow;  
    public readonly DateTime later = DateTime.Now.AddHours(2);  
  
    public SomeClass()  
    {  
 this.rightNow = DateTime.Now;  
    }  
}

So, we have two readonly fields: The first is private, the second one is public (we usually have properties for that, but bear with me here). The first is declared without a value (we can do that with readonly fields, unlike with constants), while the other one is initialized immediately. You will also notice that we're using the DateTime class as the data type, and we assign a non-constant value to it. In other words, we do a lot of stuff that we can't do with constants, making readonly fields a nice alternative to constants.

Notice how I assign a value to the rightNow field in the constructor of the SomeClass class. As already mentioned, this is the last chance to assign a value to a readonly field. After that, whether you are in a method inside the defining class or outside, you will get a compile error if you try to assign a value to a readonly field.

## **Summary**

A constant can be defined either inside the scope of a method or on the class level. It allows you to define a value which is known already at compile time and which can't be changed later on. Typical types used for constants are integers, floats, strings, and booleans. If you're looking for more flexibility, try the readonly field, as described above.

Partial Classes

If you have worked with C#, or perhaps even another programming language, you are used to the fact that the name of a class has to be unique - there cannot be two classes with the same name, unless they are in different namespaces. However, at one point, Microsoft decided to change this, with the introduction of something called partial classes.

When you define your class with the partial keyword, you or someone else is allowed to extend the functionality of your class with another class, which also needs to be declared as partial. This is useful in the following situations:

* When you have a very large class - you can then keep it in multiple files, to make it easier to work with various parts of the classes. For instance, you could have all the properties in one file and all the methods in another file, while still just having one class.
* When you work with a designer, like the one in Visual Studio - for instance with WinForms, where all the automatically generated designer code can be kept in one file, while your code is kept in another file.

Let me illustrate this with an example. In my project, I have the usual Program.cs, found in a console app. Besides that, I have added two files: PartialClass1.cs and PartialClass2.cs. Here are the files and their contents:

PartialClass1.cs

using System;  
  
namespace PartialClasses  
{  
    public partial class PartialClass  
    {  
 public void HelloWorld()  
 {  
     Console.WriteLine("Hello, world!");  
 }  
    }  
}

PartialClass2.cs

using System;  
  
namespace PartialClasses  
{  
    public partial class PartialClass  
    {  
 public void HelloUniverse()  
 {  
     Console.WriteLine("Hello, universe!");  
 }  
    }  
}

Notice that both classes are defined with the partial keyword and have the same names. Also notice that each of them define a method - HelloWorld() and HelloUniverse(). In our Program.cs we can now use this class as if it was defined in only one place, just like any other class:

using System;  
  
namespace PartialClasses  
{  
    class Program  
    {  
 static void Main(string[] args)  
 {  
     PartialClass pc = new PartialClass();  
     pc.HelloWorld();  
     pc.HelloUniverse();  
 }  
    }  
}

## **Summary**

With partial classes, you can split your classes into multiple files, either because the class definition is very large or when the tools you work with benefits from it, like with the Visual Studio designer for WinForms.

# Local functions

We learned in previous articles that methods and properties belong to classes in C#. Inside methods, you can have local variables, which are variables only accessible in the scope of this specific method. This makes sense because you'll often have temporary data you need to store, but which shouldn't be accessible from other classes or even other methods in the same class. Previously, you couldn't do the same thing with methods - if a method was declared on a class, it could, at least, be accessed from other methods within the same class, but in C# version 7, the concept of **local functions** was introduced.

A local function is declared inside an existing method and can only be accessed from this method. This encapsulates functionality very tightly and also makes it clear to any readers of your code that this functionality is only relevant for the declaring method. A local function looks like a regular method, but without the visibility modifier, since a local function is always only accessible from within the declaring method. Here is an example:

public void MethodWithLocalFunction()

{

bool doesNameStartWithUppercaseChar(string name) //this is the local funct

{

if(String.IsNullOrEmpty(name))

throw new Exception("name parameter must contain a value!");

return Char.IsUpper(name[0]);

}

List<string> names = new List<string>()

{

"john doe",

"Jane doe",

"dog Doe"

};

foreach(string name in names)

Console.WriteLine(name + " starts with uppercase char: " + doesNameStartWithUppercaseChar(name));

}

A quite silly example, but it does demonstrate how you can declare a local function (in this case called doesNameStartWithUppercaseChar) inside a method (called MethodWithLocalFunction) and then call it one or several times from inside the method.

As you can see in my example, I declare the local function in the start of the method. You are free to change this, e.g. by declaring it in the middle or the end of the method. Only in one case will it make a difference: A local function is allowed to access variables declared inside the declaring method, but only if they have been declared before the local function. So if you want to take advantage of this behavior, you will have to modify the method, e.g. like this:

public void MethodWithLocalFunction()

{

int nameMaxLength = 10;

List<string> names = new List<string>()

{

"john doe",

"Jane doe",

"dog Doe"

};

foreach(string name in names)

Console.WriteLine(name + " starts with uppercase char: " + doesNameStartWithUppercaseChar(name));

bool doesNameStartWithUppercaseChar(string name)

{

if(String.IsNullOrEmpty(name))

throw new Exception("name parameter must contain a value!");

if(name.Length > nameMaxLength)

throw new Exception("name is too long! Max length: " + nameMaxLength);

return Char.IsUpper(name[0]);

}

}

Notice how I declare the nameMaxLength variable inside the method and then access it inside the local function.

## **Static local functions**

In C# version 8, support for **static local functions** were added. As of writing, the only difference between a regular and a static local function is the fact that a static local function can't use variables from the declaring method - in other words, they no longer share scope. So if you want to make sure that your local function can't reference or change variables from the method, just declare it as static, like this:

public void MethodWithLocalStaticFunction()

{

int nameMaxLength = 10;

static bool doesNameStartWithUppercaseChar(string name)

{

// Local variables, e.g. nameMaxLength, are no longer accessible here....

if(String.IsNullOrEmpty(name))

throw new Exception("name parameter must contain a value!");

return Char.IsUpper(name[0]);

}

....

}

## **Summary**

In some situations, local functions can be a great little helper when you need to encapsulate and reuse very specific functionality. As an alternative, if the functionality can be reused from other methods, you may want to consider adding it as a real method on a helper class or as an extension method.