Object-Oriented Programming with C#

Getting Started

By Per Laursen
24-08-2025

Introduction

In this chapter, we take the first steps towards understanding what "computer programming" is all about. We introduce some of the software tools we will be using for developing C# programs, and we also take a first look at the structure of a so-called **C# project**.

The Programming Process

If you have never tried something like computer programming before, it may seem like a mysterious activity – what is it really that we are doing? If we primarily focus on computer programming as a way of defining "business logic", we are usually ***defining and manipulating a model of a small part of the world***.

What does that mean more specifically? Suppose we wish to create a computer program – or **App**[1], for short – for administration of a school. Then we probably need to store and process certain information about students (and other things) in the App. What information is relevant to know about a student? Date of birth? Shoe size? That will depend entirely on the **requirements** to the App, which somebody will have to define.

The outcome of such a requirement definition process will likely be that some parts of the available information is needed, while other parts can be left out. The "model" of a student in the App will thus only contain certain information; the information which is relevant in relation to the requirements. Exactly which information we include will depend on the specific situation, i.e. the specific requirements. Once we have figured out what information we need to include for each "concept" (student, teacher, classroom, course, etc.) in the model, we need to figure out how to represent that information in our App. We get to that part very soon!

In almost all cases, the information will need to be processed in certain ways. A very simple processing could be just to show the information to a user of the App. A more complex processing could be to change the information. For a student, some information will probably never change (e.g. the date of birth), while some information will most likely change (e.g. the number of courses taken).

The way information changes can range from quite simple to very complex. The simplest change could simply be to change the current value to a new, given value. Say, when a student has taken one more course, the number of courses is increased by one. Other changes might be more complex. If we e.g. store an average of the marks obtained for the exams passed by the student, then passing an additional exam will require a recalculation of the mark average. In any case, there will always be rules for how to process the information.

---

[1] We will in general refer to a computer program as an App (short for "Application"), without assuming anything specific about what platform the App runs on. This could be an ordinary PC, a smartphone, a tablet, or some other device.

In programming, we then try to <u>translate</u> such rules (formulated in a human – or at least human-friendly – language) to instructions/logic written in a language the computer can understand; or more precisely: written in a language that

- We as humans can use to express such rules with relative ease, and
- The computer (using specialized software called a **compiler**) can translate into a language the computer hardware understands "directly"

A wide range of such **programming languages** exist. Some are quite obscure and only known to few, while others have gained widespread popularity. The programming language named **C#** (pronounced "C-sharp") belongs to the latter category, and is used in these notes. C# is a so-called **Object-Oriented** language; other such languages are e.g. Java and C++.

So, translating our rules into the chosen programming language will result in writing a number of **statements**. A single statement usually performs a quite simple step of data processing, so most interesting programs will contain a large number of such statements (many thousands, even millions...).

When we have a large body of statements, we need to <u>organize</u> them into larger units. One such unit is a **method**, which is a fairly small collection of statements (usually less than twenty) performing a somewhat more complex kind of data processing. Collections of methods can then be organized into even larger units called **classes**, and so forth. We will discuss such units of organization later in these notes.

Software Tools

In order to write Apps using the C# language, we need some software tools to help us with this. These software tools should enable us to:

- Write C# code as easily as possible
- Write C# code of high quality
- Write C# code in collaboration with other software developers
- Help us to obtain – and maintain – an overview of the entire body of code, including all the various units of code organization
- Translate our C# code into code that the computer can run directly
- Help us find and fix errors in our code, both when the code is written (syntax errors) and executed (logic errors)

Some of these features are somewhat weakly defined (what does e.g. "high quality" mean in relation to C# code…?), but we will try to be more specific later in the notes.

Microsoft Visual Studio - overview

C# is invented by Microsoft, and their tool **Visual Studio** can on its own help us with much of the above. You can obtain a free copy of Visual Studio from the school[2].

Visual Studio is a professional and commercial tool, with a lot of "bells and whistles". It can therefore appear somewhat overwhelming at first glance. Fortunately, we only need to understand and use a fraction of the functionality:

- Understand the structure of a so-called solution or **project**
- Be able to navigate between the files in a project
- Understand the role of the files included in a project
- Add code to a project
- Compile, build and run a project
- Understand various responses from Visual Studio, like error messages, warning messages, suggestions, etc..

Before we dig into the details of how to perform the above actions in Visual Studio, we need to take a brief look at the overall structure of Visual Studio as such.

Tools, extensions and packages - overview

In the earlier days of Visual Studio, Microsoft seemed to approach the development of Visual Studio in a "monolithic" fashion, where the intention was to build a single, stand-alone tool, which (ideally) would contain everything you need in order to develop software. As the world of software development has become more and more complex and fragmented, Microsoft has now adopted a much more open strategy, where Visual Studio has the role of a sort of "functionality hub", which on its own only contains limited functionality, but instead allows you to add additional functionality to Visual Studio by different means. The three major ways to add functionality is through **tools**, **extensions** and **packages**.

---

[2] Details on how to do this may vary from course to course, and is thus found elsewhere

Tools (workloads)

The term **tool** may be a bit misleading in this context, since it actually refers to a number of so-called **components**, which add some capabilities to Visual Studio. Since a very large number of these components exist, Microsoft have bundled them into so-called **workloads**. A workload is thus a collection of a (large) number of components, which add certain capabilities to Visual Studio, for instance the ability to develop applications for mobile devices. Below is an example of some of the workloads that can be added to Visual Studio.
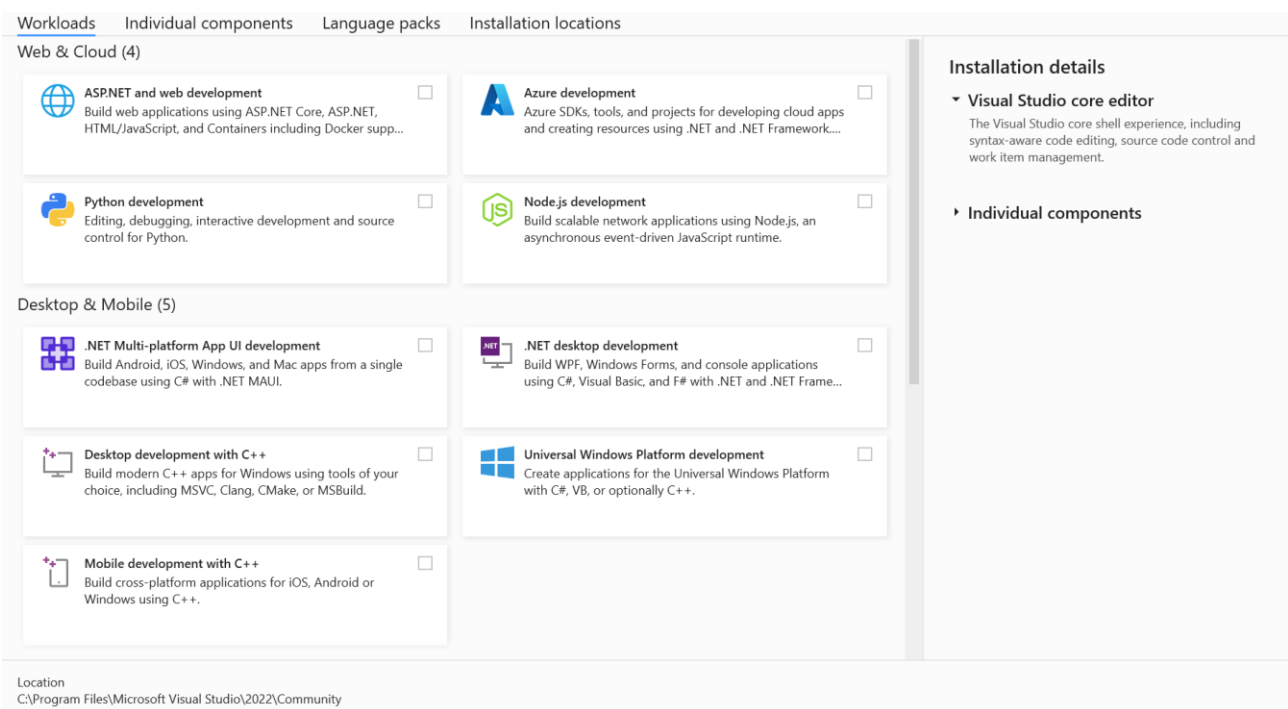
Extensions

The workloads described above are thus optional to include in your own customized setup of Visual Studio. Still, they mainly consist of tools developed by Microsoft. It is however also possible for a third-party developer to develop software that integrates into Visual Studio. By "integrates into" is meant that once the third-party software in question has been installed, it will <u>not</u> appear as an extra application, but rather manifest itself as extra functionality added to Visual Studio itself. The functionality may be very subtle; it could e.g. just add some keyboard shortcuts to the already existing shortcuts.

That fact that Microsoft has opened the door for third-party developers has resulted in an abundance of such extensions. You can (try to) get an overview of these extensions at the Visual Studio Extension Marketplace[3].

NuGet packages

The extensions described above will usually add some permanent features to Visual Studio, for instance an enhancement to the development user interface. An extension is thus <u>not</u> something that relates to a specific development project, but rather to Visual Studio seen as a tool. Suppose now that a third-party developer has come up with a nice piece of software – maybe some very efficient algorithms for data encryption – and wish to make this software available to other developers. This can be done in the form of a so-called **NuGet package**. Such a package is simply a way of distributing code from third-party developers. The ability to use NuGet packages directly from Visual Studio has been a part of Visual Studio for some years now, and seems to have evolved into the *de facto* standard for package distribution.

A very important difference between extensions and packages is that packages are added to <u>individual</u> projects! If you are developing an application which needs to use encryption, the project for that application may need to refer to a NuGet package containing that functionality. If you are developing another application which does not use encryption, it doesn't need to refer to that package. In general, you try to limit package references to those packages you actually use, to keep the size of your applications as small as possible.

What should I install?

The concept of piecing together your individual setup of Visual Studio may appear confusing at first. Don't worry too much; if you miss something along the way, you can always add it to your installation later. Also, we only need to install a fairly small set of tools and extensions to begin with.

With regards to **workloads**, we only need to have two workloads installed initially. Once you have successfully installed Visual Studio and started it, choose **Tools | Get Tools and Features** from the menu. This should bring up the overview of workloads we saw a couple of pages back:

---

[3] https://marketplace.visualstudio.com/
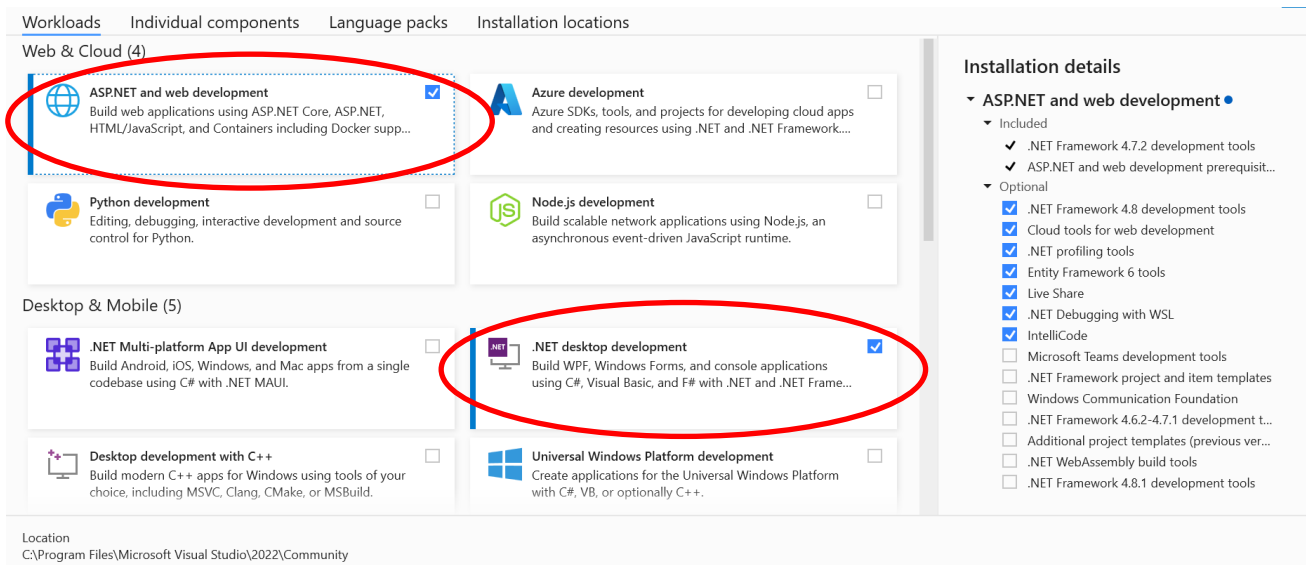
Initially[4], we only need to have these two workloads installed:

- .NET desktop development
- ASP.NET and web development

If there is already a checkmark in each of these boxes, you don't need to do anything – just close the window. If not, check the missing boxes and press **Modify**. Note that these workloads are quite large – some of them contain several Gigabytes of data – so installation may take a while. Be patient ☺. Also note that the content of a specific workload may change over time, so the details you see for e.g. the .NET desktop development workload may differ from what you see in the image above.

With regards to **extensions**, we don't need any extensions initially.

---

[4] Strictly speaking, only the .NET Desktop Development workload is needed from Day 1.

# Code organization and Visual Studio basics

We should now be *up-and-running* with Visual Studio. We will now try to open a (very small) body of code in Visual Studio, for two purposes:
- Investigating how code is organized
- Trying to load, navigate, edit and run the code

Loading code into Visual Studio

For demonstration purposes, a very small body of code named **Sandbox** has been created. This body of code should be available to you – exactly where will depend on the course, but your teacher can inform you about this ☺. With this information, you should be able to follow the below steps:

First, start Visual Studio. This will show you a window looking like this (for you, the left-hand side of the window is probably empty):
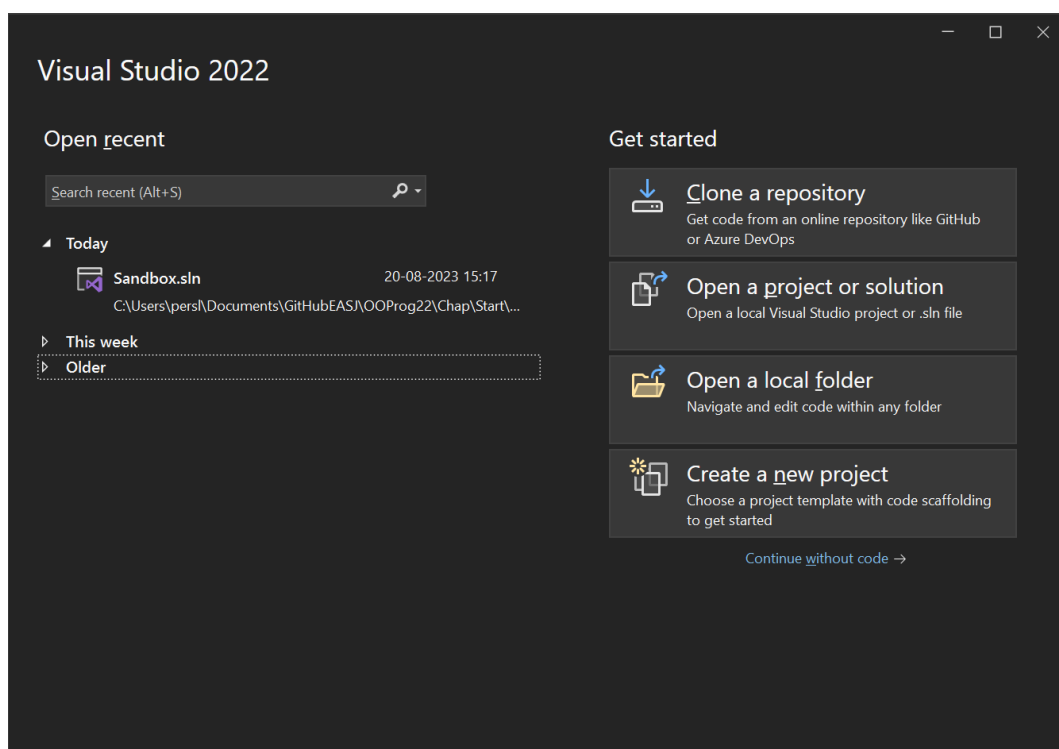

Next, choose **Open a project or solution**. This will open a file explorer window. Navigate to the folder where the **Sandbox** body of code is located (again, your course teacher will inform you of the actual location). Once you have navigated to the correct folder, you should see something like this:
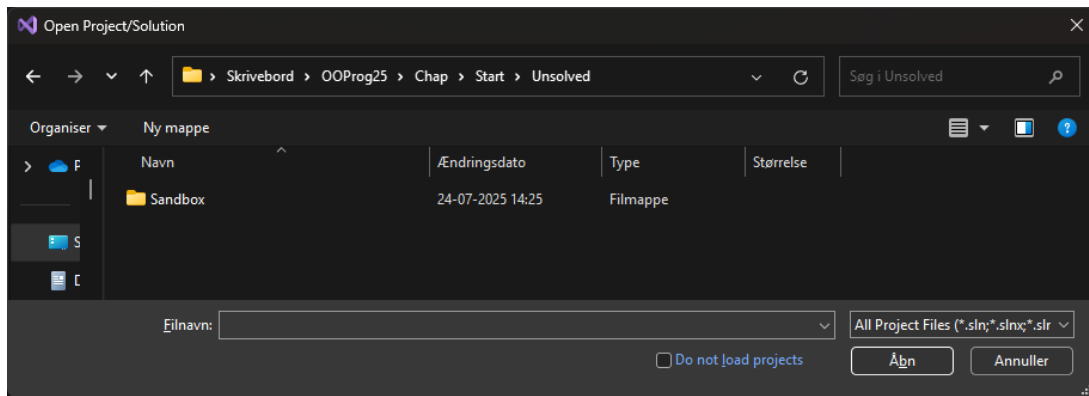
The folder named **Sandbox** will contain all of the elements which constitute a so-called Visual Studio **solution**. We will discuss what a solution is in just a moment. For now, just navigate into the folder, which should look like this:
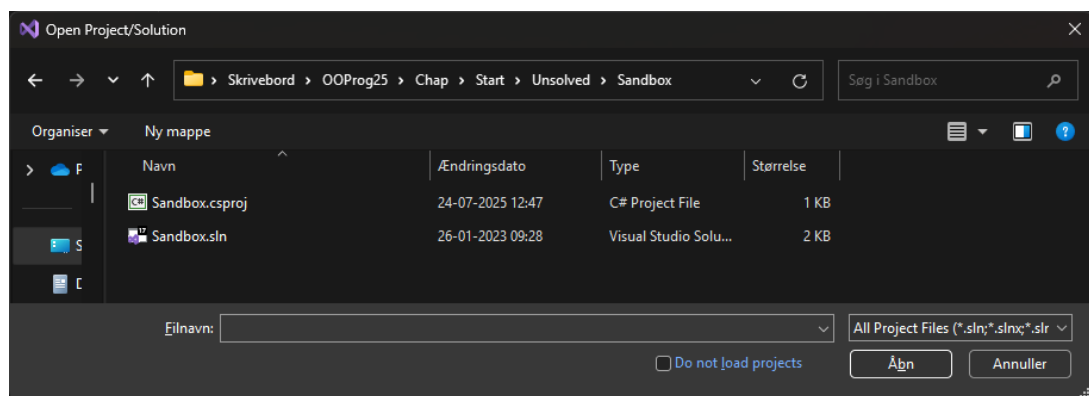
Notice the bottom file **Sandbox.sln** of type **Visual Studio Solution**. This file contains the top-level definition of the content of this particular solution. Now double-click on **Sandbox.sln** – this will load the solution into Visual Studio. After a short while, Visual Studio should look something like this:
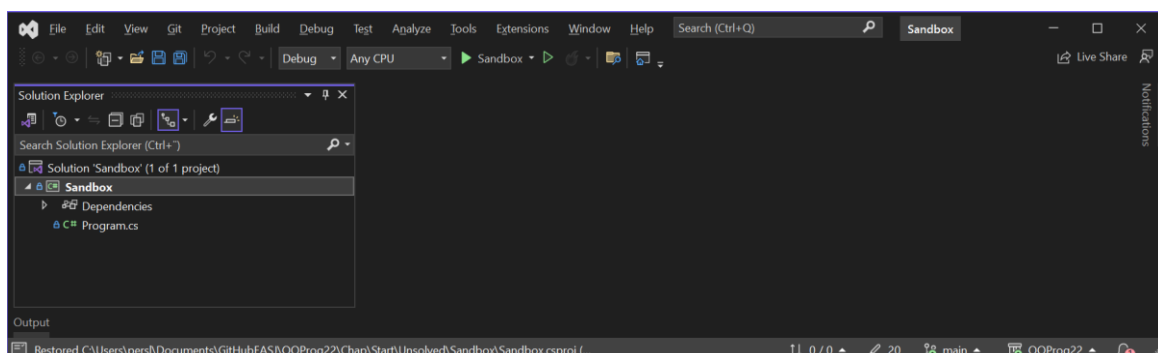
If Visual Studio doesn't look exactly like this for you, don't panic! First of all, some additional windows might be visible as well; they are not important now, so you can leave them *as-is*, or just close them. The only interesting window for now is the

window with the title **Solution Explorer**. If you for some reason don't see a window with this title, choose **View | Solution Explorer** from the menu, which will open the window again.

So, what are we seeing in this window? In order to understand that, we first need a basic understanding of code organization in Visual Studio.

Code organization

The highest unit of organization in Visual Studio is a **solution**. Remember that Visual Studio is an advanced, industrial-strength tool, which can handle very complex software development tasks. This could imply that the entire "solution" to e.g. a school administration system would contain several applications (say, a smartphone App for students, a desktop App for staff, another desktop App for administrators, etc.). All these applications should be manageable as one integrated solution. A solution can therefore consist of several **projects**.

The next level of organization is thus a **project**. A project will usually correspond to a single application. If you have several projects in a solution, you will still be able to modify, compile and run a single project, without involving the other projects.

With this information, we can already better understand what we just did, and what we see in the ***Solution*** (aha!) ***Explorer*** window.

First, we navigated to a folder named **Sandbox**. In Visual Studio, a solution is typically contained in a file folder with the same name as the solution it contains. So, this folder contains a solution called **Sandbox**.

Second, we entered the folder. The folder contained

- A file named **sandbox.sln** (a Visual Studio Solution file)
- A file named **Sandbox.csproj** (a C# project file)

Remember that a solution can contain several projects? This particular solution (named **Sandbox**) contains a single project (also named **Sandbox**). It might seem confusing that we have a project with the same name as the solution it is part of, but this is actually a quite common naming convention for small solutions that only contain a single project. It will be quite a while before we deal with solutions containing more than one project. Also, it has in this simple case been chosen to have both the .sln file (defining the entire solution) AND the .csproj file (defining a single

project) in the same folder. A alternative – which is definitely recommendable when working with multi-project solutions – is to have each project located in sub-folders to the solution folder. The subfolder corresponding to a specific project will then usually have the same name as the project itself.

It can initially be a bit confusing to distinguish between the file structure and the logical structure of a solution. The logical structure is defined by the contents of the solution and project files, plus the actual code itself. This structure always follows the same pattern: Solution -> Project -> Code. The file structure is just the physical location of the various files. This can – in principle – be chosen more freely, but the usual practice is to use a structure closely related to the logical structure, like e.g. using subfolder for projects. In order to keep things as simple as possible, we have however chosen to deviate from this general principle, when the solution only contains a single project.

With this in place, we can make more sense of the content in the *Solution Explorer* window:
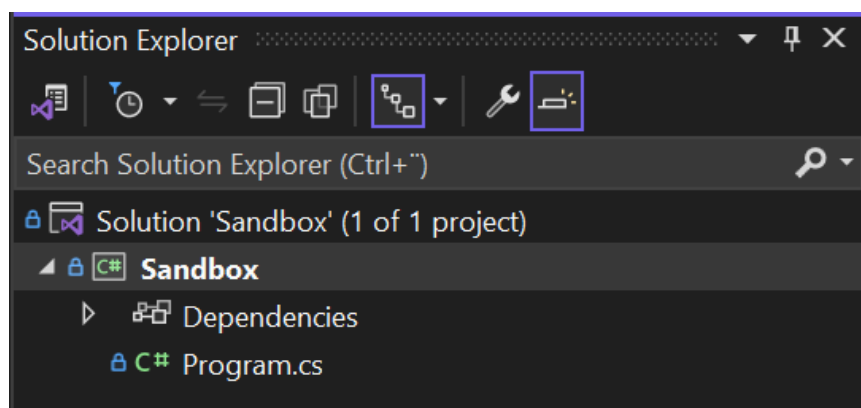
This should be read as "Visual Studio has now loaded the solution named **Sandbox**. That solution contains a single C# project named **Sandbox**".

What's does the C# project then contain? Seemingly two elements:
- An element named **Dependencies**. This element is not relevant for now.
- An element named **Program.cs**.

Program.cs contains actual C# code, so we have now reached the really interesting part. Before proceeding, we just need to briefly return to the discussion about code structure.

What is the next unit of organization below **project**? The simple answer is **classes**. Classes is where the actual C# code resides, and we will deal with classes in great detail in these notes, since they are the very foundation of Object-Oriented Programming. A class is (usually) defined in a single file with the extension **.cs**. In the **Sandbox** project, there is only a single class **Program**, defined in the file **Program.cs**.

For completeness, it should be mentioned that a unit of organization between **project** and **class** actually exist, called **namespace**. A namespace is a way to organize classes that in some sense belong together, which is useful in projects with many classes. However, if you have very few classes in your project, the most common setup is to have just one namespace, with the same name as the project. We will not really utilize namespaces before our projects grow beyond these "few classes".

Returning to classes – which we advertised as containing actual C# code – the next unit of organization is **methods**. A method is a collection of C# **statements**, that performs some useful task. We will deal quite a lot with methods later on. It should also be mentioned that classes can contain more than just methods, but in terms of code organization, they can for now be thought of as containers for methods.

With statements, we have reached the end of the line in terms of code organization, since statements are the "atoms" of code. Let us then review the entire hierarchy of code organization:

A **solution** contains a number of
    projects, that contain a number of
        namespaces, that contain a number of
            classes, that contain a number of
                methods, that contain a number of
                    statements

A six-tiered hierarchy, no less! This may seem overwhelming, but try to compare it with a publication of a large body of text, say, the collected works of Kierkegaard (a Danish philosopher of some fame). Such a publication would probably be organized like this:

A **publication** contains a number of
    volumes, that contain a number of
        chapters, that contain a number of
            sections, that contain a number of
                paragraphs, that contain a number of
                    sentences, that contain a number of
                        words

A seven-tier hierarchy… This hopefully illustrates that the code organization is as such quite meaningful, and not overly complex. However, the solutions we will encounter in relation to these notes will usually have a simpler structure:

A **solution** that contains <u>one</u>
    project, that contains <u>one</u>
        namespace, that contain <u>a few</u>
            classes, that contain <u>a few</u>
                methods, that contain <u>a few</u>
   **statements**

This also holds for the **Sandbox** solution. Going forward, we will try to work in a *bottom-up* fashion, where we initially focus entirely on writing statements, without thinking too much in terms of methods and classes.

We conclude this first brief look at Visual Studio by trying to actually run the code we have loaded into Visual Studio, i.e. the **Sandbox** project. In general, C# code must be <u>compiled</u> and <u>built</u>, before it can be <u>executed</u> (running the code is often denoted as <u>executing</u> the code). Compiling and building C# code essentially consists of two activities, both performed by Visual Studio
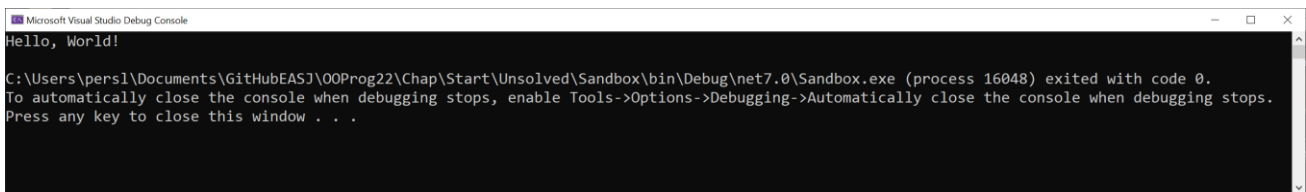
1. Checking that the code follows the syntax for the C# language
2. Translating the code into a language the computer can execute directly

The process is a bit more sophisticated than this, but we need not worry about that now. One important thing to note is that even though the code passes successfully through these steps – and can therefore be executed – there is no guarantee whatsoever that the code <u>behaves</u> as we intend it to, i.e. that it complies with our requirements. We will return to this distinction later.

The **Sandbox** project does contain code that is ready-to-run, and we set the execution of the code in motion by either pressing *F5* on the keyboard, or clicking on the button containing a small green triangle in the toolbar:
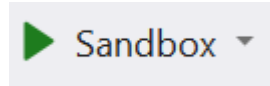
If you are able to run the code successfully, you should see something like this appear on your screen:

This somewhat primitively looking window is a so-called **console application**. This is what most Apps looked liked before computers with Windows and MacOS became mainstream. It does look rather dull, and only allows input/output in character form, but using this style initially enables us to postpone having to learn about GUI (GUI: Graphical User Interface) development. GUI development is actually a quite complex activity, and we need to know a lot of basic programming terms before it makes any sense to start learning about it.

As advertised on the screen, the App does nothing more than print *Hello, World!*, and now awaits that you press a key on the keyboard to terminate it (there is also quite a bit of "noise" between those two lines, which is not important for now). Once you do that, you have successfully loaded, compiled, built and executed your first C# application using Visual Studio!

Statements and Syntax

As stated earlier, we will investigate the organization of code in a *bottom-up* manner, starting with the simplest entity: the **statement**.

A statement can be thought of as a kind of "code atom", i.e. it is a fundamental code building block, but it also has an internal structure, which must follow certain rules. We cannot just mash up any sequence of characters and claim it to be a valid C# statement, just as we cannot throw together a random mix of electrons, neutrons and protons, and expect to end up with a stable, useful atom. Only certain combinations qualify as being valid C# statements.

More specifically, a statement is an instruction to the application about what to do next. This could be to

- Perform an arithmetic or logical calculation
- Read or write data to a file, the screen, etc..
- Control the "flow" of execution, i.e. choose between several alternatives about what to do next
- ...and a lot of other useful actions

You may already here sense that statements tend to fall into two board categories: statements that actually <u>do</u> something, like a calculation, visualization or data transfer, and statements that <u>control</u> what to do next, depending on certain conditions. We shall see several examples from both categories in these notes.

Returning to Visual Studio; if you – assuming the **Sandbox** project is still loaded – double-click on the file **Project.cs** in the *Solution Explorer* window, a new window will open, showing the content of that file:

```
Console.WriteLine("Hello, World!");
```

The text in the line is indeed a C# statement. It instructs the computer to print out the words *Hello, World!* on the screen. However, if you are new to programming in general, that may be hard to figure out just by reading the line of code... The sentence *Hello, World!* is indeed part of the line, but it is wrapped up in a lot of other stuff. What that "stuff" precisely means is not that important – we will learn to understand it later on. Also, don't worry about the small graphical symbols – like the little screwdriver symbol just left to the text – that are also visible. They appear because Visual Studio have some suggestions to us concerning the code; later on, we will also discuss how Visual Studio communicates with us.

To put the issue of understanding code in more general terms: there will usually be a "gap" between what we <u>intend</u> the computer to do, and how we <u>express</u> that intention in a programming language. In plain human language, our intention could be written as:

Please display the words ***Hello, World!*** on the screen.

In C#, that intention is expressed as:

```
Console.WriteLine("Hello, World!");
```

If somebody could make a compiler that could directly translate the human-language intention into executable code, the world wouldn't need programmers… However, human language is inherently vague and ambiguous, while a computer needs very precise instructions! If you think about it, the human-language intention leaves many questions unanswered, like e.g.

- How should the words be displayed (where on the screen, color, size, etc.)?
- What should be done after displaying the words? Stop the application, wait for the user to do something, or…?

And this is just for an extremely simple intention! For more complex intentions, we need intermediate "stops" on the road from intention to code, like requirement specifications, designs, etc..

Another difference between human language and C# code is the tolerance for errors. Suppose we made some small spelling mistake in our intention description:

Please displlay the words ***Hello, World!*** on the screen.

This small mistake would probably not hinder another human being in understanding the intention. However, a similar mistake in the C# statement, like

```
Console.WriteLline("Hello, World!");
```

will have dire consequences (Try it! Go to the file **Program.cs**, make the change, and see what happens. Also try to execute the program…). The compiler is absolutely unforgiving about errors! A C# statement has to strictly follow a predefined **syntax** for that particular type of statement. Imagine that your mail application had a similar Draconian[5] attitude towards errors, absolutely refusing to send a mail unless there are ZERO spelling and grammatical errors in the content…

---

[5] Draconian: To apply severe punishments to small offenses

The fact that C# programming (and most programming, actually) is a discipline that requires strict adherence to a given syntax, is something you need to come to terms with as an aspiring software developer. Fortunately, the software tools available now do a very good job in assisting you with getting the syntax right.

Understanding what Visual Studio is saying

If you try to modify or add code to **Program.cs** in the **Sandbox** project, you will quickly notice that Visual Studio will provide suggestions to – and even point out errors – while you type! Visual Studio's eagerness to help you can feel a bit intrusive and even confusing at first, but once you get used to it, you will likely find it quite helpful.

If you tried to do the small change suggested above, you probably noticed that as soon as you made the change, something happened in the code editor window:
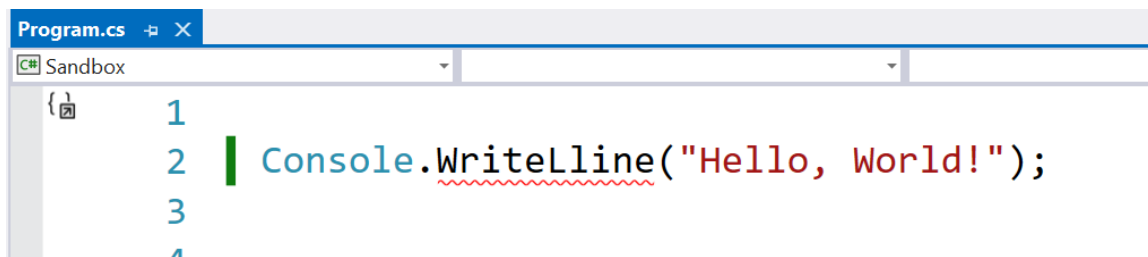
A wavy red line – also known as a "squiggly" line – appeared just below the word **WriteLline**. This is Visual Studio telling us that there is an error in the statement, and the error is more specifically related to the word **WriteLline**. If you hover the mouse cursor over the red line, Visual Studio will provide more details:
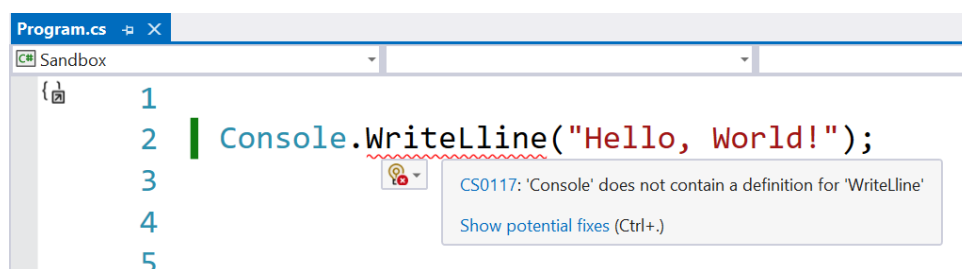

These details can initially feel somewhat cryptic, but the essence of the message is in this case that we misspelled **WriteLine**. Visual Studio is actually also clever enough to suggest how the error could be fixed. If you click on the small triangle next to the lightbulb, these suggestions will appear:
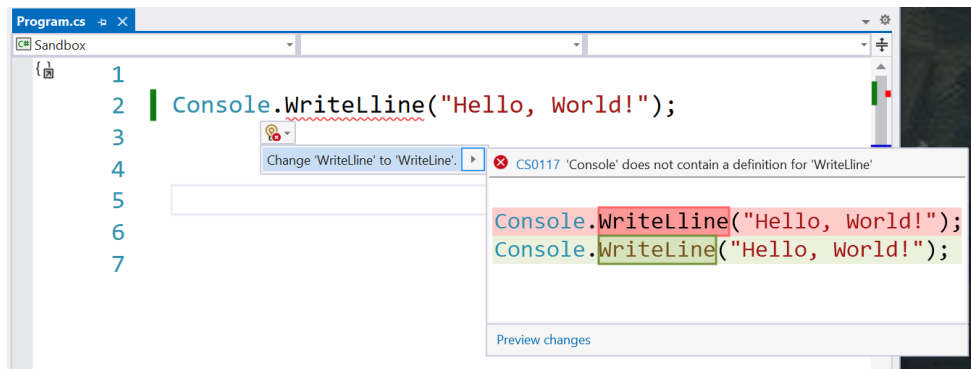
In this case, there is only one suggestion: Change **WriteLline** to **WriteLine**. Next to the suggestion itself, Visual Studio shows a "preview" of the change. If you are happy with the suggestion, you can simply select it, and Visual Studio will apply the change to the code. Neat!

In general, you should try to <u>fix errors as soon as you see them</u>! If you have more than one error in your code, some errors may actually not be true errors, but rather errors that occur because the previous code is incorrect. In that case, you should try to fix the errors from the top and downwards.

In addition to being eager to point out errors and give suggestions to existing code, Visual Studio is also quite eager to help you write new code. In **Program.cs**, try placing the mouse cursor just after the ";" at the end of the statement, and press *Return* on the keyboard. Visual Studio then promptly comes up with a suggestion for code to put on that line:
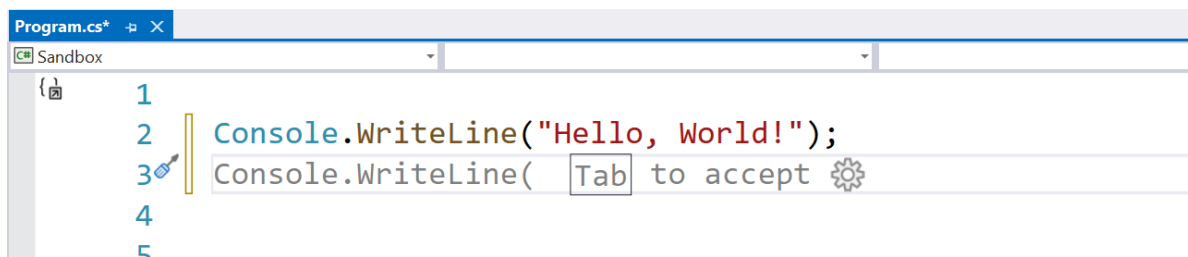
Note that this is indeed only a suggestion! As stated, you have to press *Tab* to accept the suggestion, which will then be inserted. If you start typing something else, the suggestion will go away. Also note that as soon as you start typing something, Visual Studio will also quite quickly produce a drop-down box with suggestions to choose from. If you e.g. type "i" in the new line, you will see something like this:
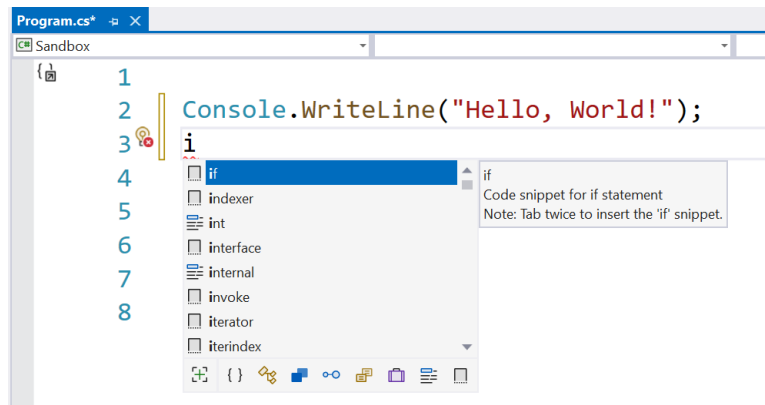
There are several other forms of productivity "enhancers" in Visual Studio, some more useful than others. We will not explicitly describe any more here, since this is by itself a quite large topic. You will probably pick many of these up along the way, as you get more hands-on experience with Visual Studio. Finally, it should also be mentioned that many of these features have settings you can play around with, in order to help you fine-tune Visual Studio to be just as helpful as you want.

Having a development environment that is so "alive" is a huge benefit, and once you get used to it, it will increase your productivity considerably. Still, it does take some time to get used to, and the error messages and suggestions are sometimes hard to decipher. Practice is the only way forward here!
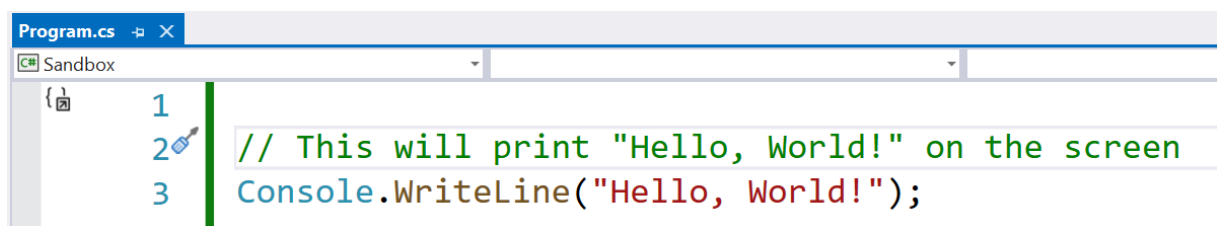
An aside on AI in Visual Studio

At the time of writing of the current version of these notes (August 2025), the usage of **AI** (Artificial Intelligence) is rapidly permeating all kinds of software development tools, and Visual Studio is no exception. For Visual Studio, this comes in the form of the **GitHub CoPilot** functionality. If you have just started Visual Studio for the first time, you may see a GitHub CoPilot icon in the toolbar. The advent of AI in programming is an extremely interesting and even somewhat controversial topic, but also a very new and quite "fluent" topic, where things are moving quite fast. We have decided not to address this topic further in (this version of) these notes; this is not to be taken as a stance against using AI-related tools as such, but rather because we want to focus on learning programming by actually crafting code ourselves. Admittedly, this is a somewhat grey area, since the auto-completion feature described earlier can also be viewed as a sort of helpful AI-like feature, but this is where we draw the line for now. But don't take this as a dogmatic statement about not using AI-based tools at all. Feel free to experiment with tools like this on your own, but make sure that you always understand the code that is being produced, using AI-tools or not.

Comments in code

We are at the brink of being able to actually use Visual Studio to study some actual C# code, and also to write code ourselves. We will initially see very small examples of C# code, which are (hopefully) almost self-explanatory. Still, code doesn't have to be particularly complex before it becomes more challenging to make sense of it. One way of making it less challenging to understand code is to add so-called **comments** to the code.

A comment is in this context a relatively small piece of text, inserted directly into the code itself. For our running example, adding a comment could look like this:

```
Program.cs
C# Sandbox
1
2  // This will print "Hello, World!" on the screen
3  Console.WriteLine("Hello, World!");
```

The green text in line 2 is a comment. The **//** symbols at the start of the line are crucial, since they indicate to Visual Studio that whatever follows on this line is to be considered a comment. A comment has <u>no</u> influence whatsoever on the actual code, and all comments are completely ignored when Visual Studio builds and runs your application. They are thus only present in order to help us human beings understand the code.

A single-line comment like the above must always start with the symbol **//**. It is also possible to write comments than span several lines. In that case, you must start the comment with the symbol **/***, and end it with ***/**. Feel free to experiment a bit with comments in **Program.cs**.

The ability to add comments to code is very common in programming languages, and it is considered good practice to add comments to code that has a certain complexity, to help those that might work with the code at a later time. However, one should be thoughtful about using comments. A first good rule is to avoid trivial comments. The comment we just added to **Program.cs** is actually an example of a bad comment, since the C# statement itself is rather trivial. Comments should provide real value to the reader. So, suppose instead we have created some really complex code. This must be a relevant place to add some comments, yes? It probably is… but it should also make you think a bit about <u>why</u> you feel that comments are needed. Maybe you have created code that is overly complex, so the urge to add comments is perhaps a

sign that the code itself needs an overhaul…? We will discuss such considerations in much more detail later in these notes. For now, just take note of the fact that it is indeed possible to add comments to C# code directly in the code itself.

Exercises

Exercise	Start.1
Project	Sandbox
Purpose	Reality check – get Visual Studio up and running
Description	The **Sandbox** project is as simple as it gets – we will just use it to verify that your installation of Visual Studio is up and running
Steps	1. Load, compile and run the project. 2. Verify that the message *Hello, World!* Is printed on the screen.