|  |
| --- |
| **EASJ Notes** |
| Object-Oriented Pro-gramming with C# |
| Getting Started |

|  |
| --- |
| By Per Laursen  01-07-2018 |

[Introduction 2](#_Toc517259189)

[The Programming Process 3](#_Toc517259190)

[Software Tools 4](#_Toc517259191)

[Microsoft Visual Studio - overview 5](#_Toc517259192)

[Tools, extensions and packages - overview 5](#_Toc517259193)

[Tools (workloads) 6](#_Toc517259194)

[Extensions 6](#_Toc517259195)

[NuGet packages 7](#_Toc517259196)

[What should I install? 7](#_Toc517259197)

[ReSharper 9](#_Toc517259198)

[GitHub 10](#_Toc517259199)

[Code organisation and Visual Studio basics 11](#_Toc517259200)

[Loading code into Visual Studio 11](#_Toc517259201)

[Code organisation 12](#_Toc517259202)

[Statements and Syntax 16](#_Toc517259203)

[Exercises 20](#_Toc517259204)

[Start.1 20](#_Toc517259205)

# 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 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 piece of the world***.

What does that mean more specifically? Suppose we wish to create a computer pro­gram – or **App[[1]](#footnote-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, …) 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 a certain way. 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 are 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 rulesfor how to process the information.

In programming, we then try to translatesuch 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 specialised 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 called **C#** (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 organise 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 organised into even larger units called **classes**, and so forth. We will discuss such units of organisation 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 developers
* Help us to obtain – and maintain – an overview of the entire body of code, including all the various units of code organisation
* 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 fluffy (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]](#footnote-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 through 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 error messages from Visual Studio

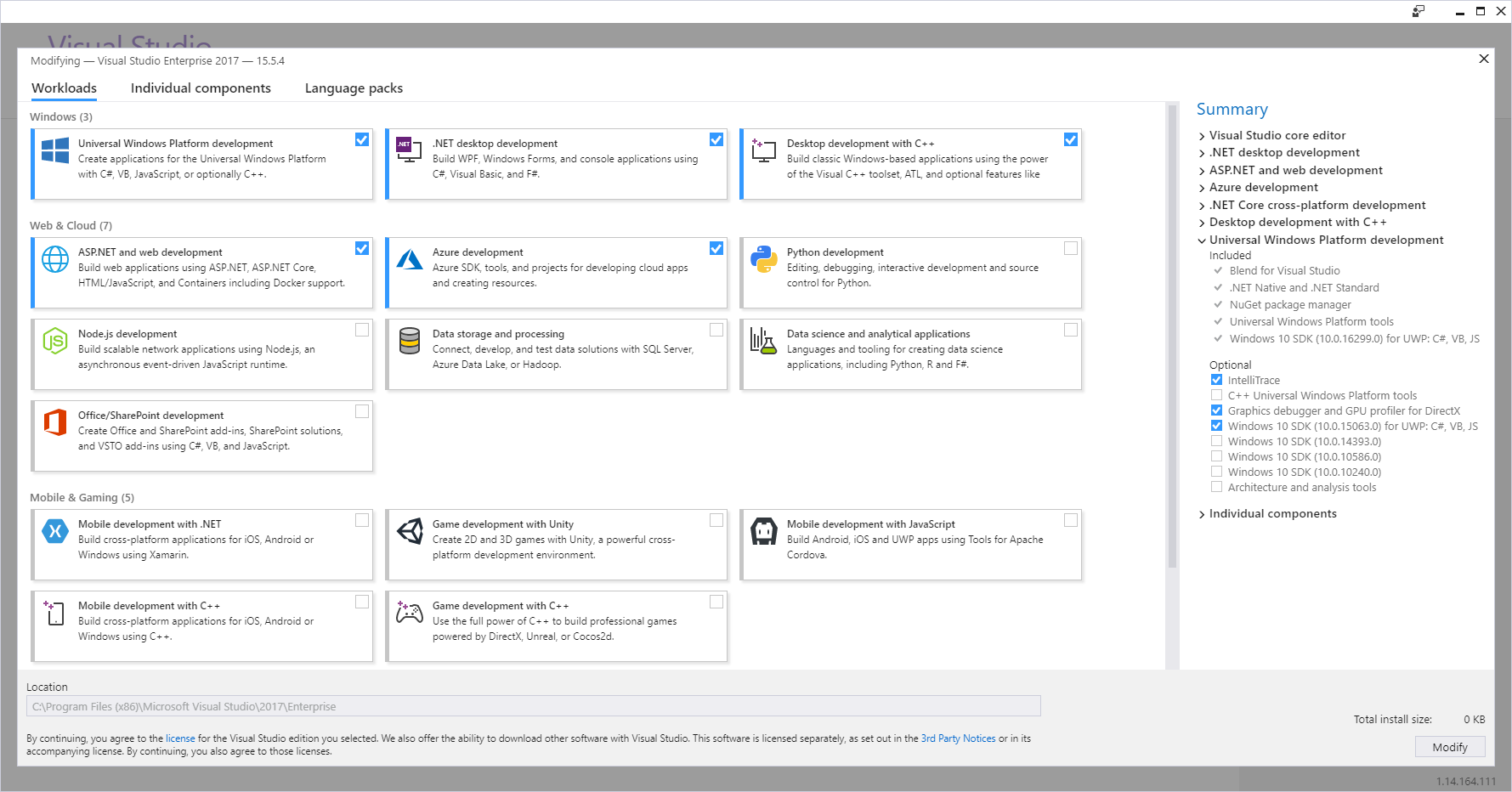
Before we dig into the details about how to perform the above actions in Visual Studio, we need to take a brief look at the overall structure of Visual Studio as such.

## 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 everyting 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**.

## 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 adds some capability 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 adds a capability 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 (also see[[3]](#footnote-3))



## Extensions

The workloads described above are thus optional to include in your own customised 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 not appear as an extra application, but rather ma­ni­fest itself as extra functionality added to Visual Studio itself. The functionality may be very “discreet”; 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 a flurry of such extensions. You can (try to) get an overview of these extension at the Visual Studio Extension Marketplace[[4]](#footnote-4).

## NuGet packages

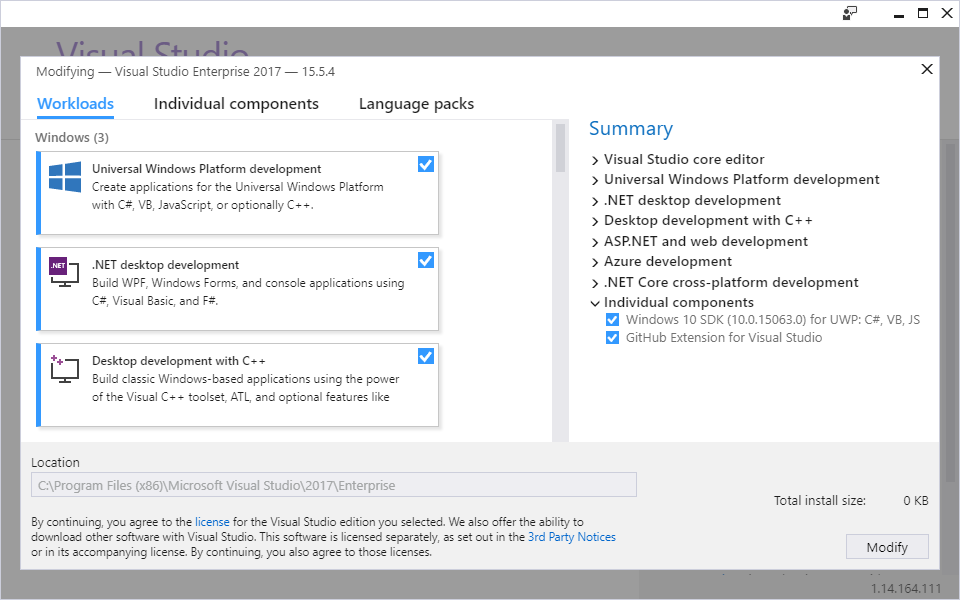
The extensions described above will usually add some permanent features to Visual Studio, for instance an enhancement to the development user interface. An exten­sion is thus not something that relates to a specific development project, but rather to Visual Studio seen as a tool. Suppose now that a third-party 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 **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 individual 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 references to packages 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 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 need to have three 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:



Initially, we only need to have these three workloads installed:

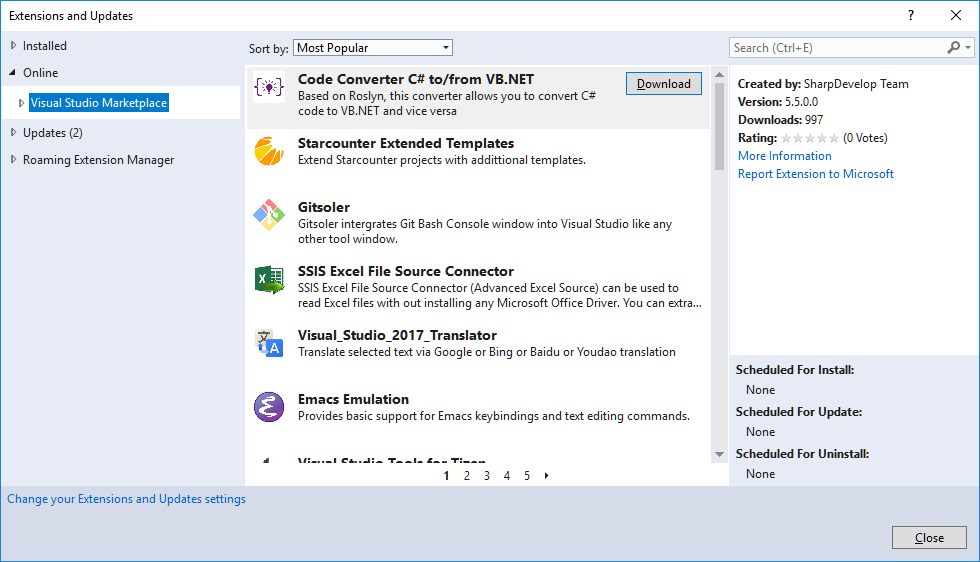
* Universal Windows Platform development
* .NET Desktop development
* Desktop development with C++

If there is already a checkmark in each of these three boxes, you don’t need to do anything – just close the window. If not, then check the missing boxes and press **Modify**. Note that these workloads are quite large – some of them several Gigabyte of data – so installation may take a while. Be patient ☺.

With regards to **extensions**, we need to install two extensions initially:

* Resharper Ultimate (by Jetbrains)
* GitHub Extension for Visual Studio (by GitHub)

We will provide brief descriptions of these extensions in a moment. The easiest way to install extension is to choose **Tools | Extensions and Updates** from the menu. This should bring up the below dialog:



If you expand the **Online** node, and highlight **Visual Studio Marketplace**, you can then use the search box in the upper-right corner to find each extension. Once found, click on the **Download** button for the extension. This will download the extension and pre­pare it for installation. The installation will however not be completed before you close Visual Studio. Once you open Visual Studio again, the extension should be installed.

## ReSharper

The ReSharper extension provides a lot of extra/improved functionality to Visual Studio, but it will not as such be visible; it will just seem like Visual Studio has been updated with extra functionality. In fact, it can be hard to distinguish between native Visual Studio functionality and Resharper functionality, but that is more or less the point. For a newcomer to Visual Studio and/or programming in general, the most note­worthy feature of Resharper is probably its ability to provide suggestions for how to fix problems in your code. This is to be understood in a quite broad sense; Reshar­per can provide suggestions for fixing errors that makes your program unable to compile (so-called **syntax errors**), but can also provide suggestions for changing code that is actually compilable, but may contain an unsafe construction, be ineffi­cient, etc.. It can initially be a bit difficult to navigate through all these suggestions, but there are (fornunately) also ways to configure how “intrusive” Resharper should be with regards to presenting suggestions. We will revisit this feature – and some other features – of Resharper later in these notes.

## GitHub

In the beginning of the chapter, we mentioned that our development tool should also enable us to collaborate with others when writing code. This is probably not a feature you will need until you embark on creating a larger App as part of a group project, but it is still useful to set up from the beginning, since you can also use it to safely experiment with code changes, without risking to lose previous work.

A general challenge in multi-developer projects is to ensure that the deve­lopers can work independently – and simultaneously – on the project. As long as the deve­lop­ers work on non-overlapping parts of the project, the process is fairly easy to manage. In reality, you may however often need to work on parts that do overlap, and you then need to be very careful about how to manage changes to the project.

This is obviously not a new problem, so the concept of **version control sys­tems (VCS)** is well-established in software development. These systems typically maintain a “master” version of the project at a central location (i.e. a location all deve­lopers can access – this is often called a **repository**), and pro­vide the deve­lopers with tools to manage changes to the project in a safe and systematic way. The exact strategy for how to manage changes may vary from system to system, and may also depend on the policies for change manage­ment agreed upon by the developers. Almost all modern VCSs also offer a lot of useful features like maintaining a complete history the project – making it possible to “roll back” a project to a certain point in time – statistics, integration with tools like Visual Studio, etc..

A lot of VCSs exist today, and it may be hard to choose one over another. The VCS called **Git** has gained widespread popularity, and we have chosen to use Git as the VCS of choice as well. The GitHub extension integrates VCS functionality based on Git into Visual Studio. But why is the extension then called **GitHub**? The term GitHub is strictly speaking a reference to the website [www.github.com](http://www.github.com), which offers web-based hosting of repositories, including – but not limited to – source code reposi­to­ries. The service uses Git technology under-the-covers, but also adds functionality on top of the native Git functionality. GitHub has also gained substantial popularity in recent years, and offers free repositories without significant limitations, as long as the repositories are public.

The GitHub extension simply adds GitHub-based VCS functionality to Visual Studio. This means that project files can be managed (w.r.t. version control) directly in Visual Studio. Several other features also become available, which we will discuss later in these notes.

# Code organisation and Visual Studio basics

We should now be up-and-running with Visual Studio, including the workloads and extensions covered in the previous section. We will now try to open a (very small) piece of code in Visual Studio, for two purposes:

* Investigating how code is organised
* Trying to load, navigate, edit and run the code

## Loading code into Visual Studio

For demonstration purposes, a small piece of code called **Sandbox** has been created. This piece 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:

1. Start Visual Studio
2. From the menu, choose **File | Open | Project/Solution…**
3. Navigate to the folder called **Sandbox**, and go into the folder
4. Double-click on the file called **Sandbox.sln**
5. The code should now load – you will see some files in the window with the title ***Solution Explorer***

If you have completed the above steps successfully, you should see something similar to this in the ***Solution Explorer*** window (if you for some reason don’t see a window with this title, choose **View | Solution Explorer** from the menu):



What are we seeing here? In order to understand that, we need a basic under­stand­ing of **code organisation** in Visual Studio.

## Code organisation

The highest unit of organisation 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.

The next level is 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 called **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 called **sandbox.sln**
* A folder called **Sandbox**

A file with the extension **.sln** is interpreted by Visual Studio as a **solution file** (.sln = solution) which holds information about a particular solution. The subfolders in the folder contains those projects that are part of the solution. So, the solution **Sandbox** thus contains a project called… **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.

Third, we double-clicked on the **sandbox.sln** file. This prompted Visual Studio to load in the **Sandbox** solution, which only contains the **Sandbox** project. This is what we see in the ***Solution Explorer*** window:



The project itself is where the really interesting stuff is. The stuff “hanging” under the **Sandbox** project in the ***Solution Explorer*** window is where the actual C# code is. Before dealing with actual code, we return again to the code structure discussion.

What is the next unit of organisation 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. A class is (usually) defined in a single file with the extension **.cs**. In the **Sandbox** project, there are thus two classes **Program** and **InsertCode­Here**, defined in the files **Program.cs** and **InsertCodeHere.cs**, respectively.

With this information, we should be able to understand almost everything we see in the ***Solution Explorer*** window. The two items called **Properties** and **References** are not interesting at this point, so they will remain a mystery…

For completeness, it should be mentioned that a unit of organisation between **pro­ject** and **class** actually exist, called **namespace**. A namespace is a way to organise 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 utilise namespaces before our projects grow beyond these “few classes”.

Returning to classes – which we advertised as containing actual C# code – the next unit of organisation 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 organisation, they can for now be thought of as containers for methods.

With statements, we have reached the end of the line of terms of code organisation, since statements are the “atoms” of code. Let us then review the entire hierarchy of code organisation:

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 organised 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 organisation is as such quite meaningful, and not overly complex. However, the solutions we will encounter in relation to these notes will usually have a rather simple structure:

A **solution** that contains one

**project**, that contains one

**namespace**, that contain a few

**classes**, that contain a few

**methods**, that contain a few

**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 compiled and built, before it can be executed (running the code is often denoted as executing the code). Compiling and building C# code essentially consists of two activi­ties, both performed by Visual Studio

1. Checking that the code obeys 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 behaves 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 Windows and Macs became mainstream. It does look rather dull, and only allows input/output in character form, but using this style first enables us to postpone having to learn about GUI (GUI: Graphical User Interface) development initially.

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. Once you do that, you have successfully loaded, compiled, built and executed your first C# appli­cation using Visual Studio!

## Statements and Syntax

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

A statement can be thought of as a “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 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
* Etc.

You may already here sense that statements tend to fall into two board categories: statements that actually do something, like a calculation, visualisation or data trans­fer, and state­ments that control 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 **InsertCodeHere.cs** in the ***Solution Explorer*** window, a new window will open, showing the content of that file (there may be some small differences between the code below, and what you actually see in the window):

**using System;**

**namespace Sandbox**

**{**

**public class InsertCodeHere**

**{**

**public void MyCode()**

**{**

**// The FIRST line of code should be BELOW this line**

**Console.WriteLine("Hello world!");**

**// The LAST line of code should be ABOVE this line**

**}**

**}**

**}**

This may look a bit overwhelming, with a lot of colors and odd characters posi­tioned strangely on the screen; for now, try to apply “tunnel vision”, and focus on a single of these lines:

**Console.WriteLine("Hello world!");**

This 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.

To put it in more general terms: there will usually be a “gap” between what we intend the computer to do, and how we express that intention in a programming language. In 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 “words” exactly. Only **Hello world!** or maybe **Hello world! on** or…?
* 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 distinction between human-language intentions 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 catastrophic consequences (Try it! Open the file **InsertCodeHere.cs**, make the change, and try to run the application…). 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]](#footnote-5) attitude towards errors, absolutely refusing to send a mail unless there are ZERO spelling and grammatical errors in the content…

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 programmer. Fortunately, the software tools available now do a very good job in assisting you with getting the syntax right. If you try to modify or add code to 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 confusing at first, but once you get used to it, you will find it quite helpful.

If you tried to do the small change suggested above, you probably noticed that a squiggly red line appeared below the incorrect piece of code. Such lines imply that something is wrong with the code! If you hover the mouse cursor over the line, an error mes­sage tooltip will pop up. Some error mes­sages may be hard to understand for a novice programmer, and the best advice is often simply to try to read the code again, and use your common sense to spot the problem.

In general, you should try to fix errors as soon as you see them! 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.

If we zoom out just a little bit in the code in the **InsertCodeHere.cs** file, you will notice some more lines of text:

**// The FIRST line of code should be BELOW this line**

**Console.WriteLine("Hello, world");**

**// The LAST line of code should be ABOVE this line**

Are these new lines also C# code? No, they are so-called **comments**. A comment is just a piece of text that Visual Studio does not consider to be code, and it will there­fore ignore it when compiling the code. Comments are thus only present in order to help the human beings working with the code. A single-line comment like the two above must always start with the symbol **//**. A multi-line comment must start with the symbol **/\***, and end with **\*/**.

The ability to add comments to code is very common in programming languages, and it is consider good practice to add comments to code that has a certain complexity[[6]](#footnote-6), to help those that might work with the code at a later time.

In this specific situation, the two comment lines have been added to outline the “sandbox” (hence the project name) within which we will play around for a while, to learn about the fundamentals of programming.

# Exercises

|  |  |
| --- | --- |
| **Exercise** | Start.1 |
| **Project** | Sandbox |
| **Purpose** | Reality check – 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. |

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. [↑](#footnote-ref-1)
2. Details on how to do this may vary from course to course, and is thus found elsewhere [↑](#footnote-ref-2)
3. https://www.visualstudio.com/vs/visual-studio-workloads/ [↑](#footnote-ref-3)
4. https://marketplace.visualstudio.com/ [↑](#footnote-ref-4)
5. Draconian: To apply severe punishments to small offenses [↑](#footnote-ref-5)
6. One could, however, also claim that if you feel the need to add comments to your code, it might be an indication that your code should be restructured to make it clearer… [↑](#footnote-ref-6)