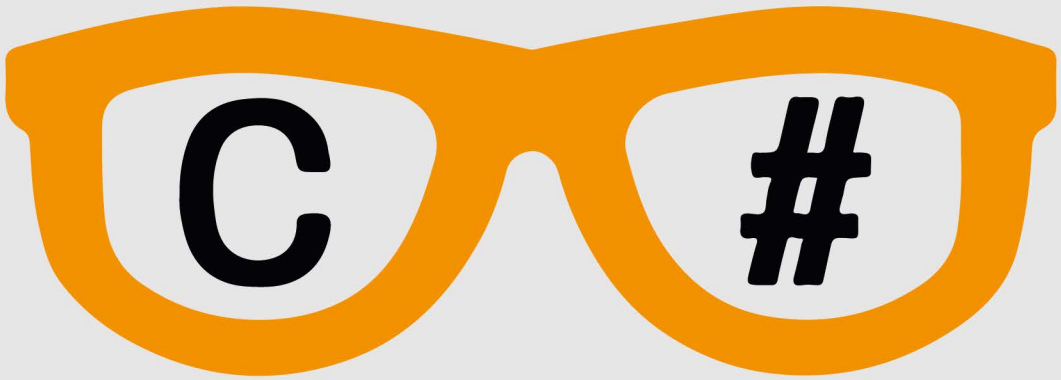


Visual Studio 2022 editie



SAMPLE

Zie Scherp Scherper

Leren programmeren in C#
van beginner naar gevorderde

DOOR TIM DAMS

Derde editie

Zie Scherp Scherper

3e editie

Object georiënteerd programmeren met C#, van beginner naar gevorderde

Tim Dams

Zie Scherp Scherper
3e editie

Tim Dams

ISBN 9789464651560

© 2021 - 2024 Tim Dams

Inhoudsopgave

| | |
|--|-----------|
| Welkom | i |
| 1 Wat je kunt verwachten | ii |
| 2 Over de bronnen | ii |
| 3 Benodigdheden | ii |
| 4 Dankwoord | iii |
| | |
| 1 De eerste stappen | 1 |
| 1.1 Wat is programmeren? | 1 |
| 1.2 Kennismaken met C# en Visual Studio | 6 |
| 1.3 Console-applicaties | 14 |
| 1.4 Fouten oplossen | 25 |
| 1.5 Kleuren in console | 29 |
| 1.6 Waar zijn de oefeningen?! | 31 |
| | |
| 2 De basisconcepten van C# | 33 |
| 2.1 Keywords: de woordenschat | 34 |
| 2.2 Variabelen, identificers en naamgeving | 35 |
| 2.3 Commentaar | 38 |
| 2.4 Datatypes | 39 |
| 2.5 Variabelen | 45 |
| 2.6 Expressies en operators | 51 |
| 2.7 Expressiedatatypes | 54 |
| 2.8 Solutions en projecten | 58 |
| | |
| 3 Tekst gebruiken in code | 67 |
| 3.1 Tekst datatypes | 68 |
| 3.2 Escape characters | 70 |
| 3.3 Strings samenvoegen | 75 |
| 3.4 Optellen van char variabelen | 79 |
| 3.5 Vreemde tekens in console tonen | 80 |
| 3.6 Environment bibliotheek | 83 |
| | |
| 4 Werken met data | 85 |
| 4.1 Appelen en peren | 85 |
| 4.2 Casting | 86 |

| | | |
|----------|--|------------|
| 4.3 | Conversie | 90 |
| 4.4 | Parsing | 91 |
| 4.5 | Invoer van de gebruiker verwerken | 92 |
| 4.6 | Berekeningen met System.Math | 94 |
| 4.7 | Random getallen genereren | 97 |
| 4.8 | Debuggen | 100 |
| 5 | Beslissingen | 105 |
| 5.1 | Relationele en logische operators | 106 |
| 5.2 | If | 110 |
| 5.3 | Scope van variabelen | 119 |
| 5.4 | Switch | 121 |
| 5.5 | Enum | 124 |
| 6 | Loops | 131 |
| 6.1 | Soorten loops | 131 |
| 6.2 | While | 133 |
| 6.3 | Do while | 136 |
| 6.4 | For-loops | 139 |
| 6.5 | Nested loops | 143 |
| 7 | Methoden | 145 |
| 7.1 | Werking van methoden | 146 |
| 7.2 | Returntypes van methoden | 149 |
| 7.3 | Een uitgewerkte methode | 151 |
| 7.4 | Parameters doorgeven | 153 |
| 7.5 | Bestaande methoden en bibliotheken | 163 |
| 7.6 | Geavanceerde methode-technieken | 167 |
| 8 | Arrays | 173 |
| 8.1 | Nut van arrays | 173 |
| 8.2 | Werken met arrays | 176 |
| 8.3 | Geheugengebruik bij arrays | 186 |
| 8.4 | System.Array | 190 |
| 8.5 | Algoritmes en arrays | 193 |
| 8.6 | String en arrays | 195 |
| 8.7 | Methoden en arrays | 199 |
| 8.8 | Meer-dimensionale arrays | 204 |
| 9 | Object georiënteerd programmeren | 211 |
| 9.1 | C# is OO in hart en nieren | 212 |
| 9.2 | Klassen en objecten | 219 |
| 9.3 | OOP in C# | 224 |

| | | |
|-----------|--|------------|
| 9.4 | Properties | 238 |
| 9.5 | OO P in de praktijk : DateTime | 252 |
| 10 | Geheugen- en codebeheer | 259 |
| 10.1 | Geheugenbeheer in C# | 259 |
| 10.2 | Objecten en methoden | 270 |
| 10.3 | Object referenties en null | 273 |
| 10.4 | Namespaces en using | 277 |
| 10.5 | Exception handling | 280 |
| 11 | Gevorderde klasseconcepten | 289 |
| 11.1 | Constructors | 289 |
| 11.2 | Object initializer syntax | 301 |
| 11.3 | required properties | 303 |
| 11.4 | Static | 304 |
| 12 | Arrays en klassen | 315 |
| 12.1 | Arrays van objecten aanmaken | 315 |
| 12.2 | List collectie | 319 |
| 12.3 | Foreach loops | 324 |
| 12.4 | Het var keyword | 326 |
| 12.5 | var en foreach | 326 |
| 12.6 | Nuttige collectie-klassen | 327 |
| 13 | Overerving | 331 |
| 13.1 | Wat is overerving | 332 |
| 13.2 | Overerving in C# | 334 |
| 13.3 | Constructors bij overerving | 339 |
| 13.4 | Virtual en Override | 344 |
| 13.5 | Het base keyword | 347 |
| 14 | Gevorderde overervingsconcepten | 351 |
| 14.1 | System.Object | 351 |
| 14.2 | Abstracte klassen | 357 |
| 14.3 | Zelf exceptions maken | 363 |
| 15 | Associaties | 365 |
| 15.1 | Heeft een-relatie | 365 |
| 15.2 | “Heeft meerdere”- relatie | 372 |
| 15.3 | Associatie of overerving? | 374 |
| 15.4 | Het this keyword | 375 |
| 16 | Polymorfisme | 379 |
| 16.1 | De “is een”-relatie in actie | 379 |

| | | |
|------------------|--|------------|
| 16.2 | Objecten en polymorfisme | 381 |
| 16.3 | Arrays en polymorfisme | 381 |
| 16.4 | Polymorfisme in de praktijk | 383 |
| 16.5 | De <code>is</code> en <code>as</code> keywords | 386 |
| 16.6 | <code>Is</code> , <code>as</code> en polymorfisme: een krachtige bende | 389 |
| 17 | Interfaces | 391 |
| 17.1 | Interfaces en klassen | 394 |
| 17.2 | Het <code>is</code> keyword met interfaces | 397 |
| 17.3 | Interfaces in de praktijk | 400 |
| 17.4 | Bestaande interfaces in .NET | 402 |
| 17.5 | Alles samen : Polymorfisme, interfaces en <code>is/as</code> | 406 |
| 18 | Bestandsverwerking | 411 |
| 18.1 | Bestands- en folderlocaties | 412 |
| 18.2 | Schrijven en lezen | 417 |
| 18.3 | Binaire bestanden | 419 |
| 18.4 | De <code>FileInfo</code> klasse | 424 |
| 18.5 | <code>DirectoryInfo</code> klasse | 426 |
| 18.6 | Klassen serialiseren | 429 |
| 19 | Conclusie | 437 |
| 19.1 | En nu? Ken ik nu alles van C#/.NET ? | 438 |
| Appendix: | Handig om weten | 439 |
| 1 | Visual Studio snippets | 439 |
| 2 | Regions | 440 |
| 3 | <code>String.Format()</code> | 441 |
| 4 | <code>out</code> en <code>ref</code> keywords | 442 |
| 5 | Foute invoer opvangen met <code>TryParse</code> | 443 |
| 6 | Operator overloading | 445 |
| 7 | Expression bodied members | 447 |
| 8 | Generics | 449 |
| 9 | Records & structs | 453 |

Welkom

Zo, je hebt besloten om C# te leren? Je bent hier aan het juiste adres. Dit boek is ontstaan als handboek voor de opleidingen professionele bachelor elektronica-ict en toegepaste informatica van de AP Hogeschool. Ondertussen wordt het ook in tal van andere hogescholen en middelbare scholen gebruikt. Ik ga je op een laagdrempelige manier leren programmeren in C#, waarbij geen voorkennis vereist is.

Eerst zullen we de fundering leggen en zaken behandelen zoals variabelen, loops methoden en arrays. Vervolgens zal de wonderlijke wereld van het *object georiënteerd programmeren* uit de doeken gedaan worden.

Je vraagt je misschien af hoe up-to-date dit boek is? Wel, het is origineel samengesteld tijdens de lockdowns in 2020... Mmm, het jaar 2020 als kwaliteitslabel gebruiken is een beetje zoals zeggen dat je wijn maakt met rioolwater. Toen eind 2021 een nieuwe versie van Visual Studio verscheen werd het tijd om dit boek grondig te updaten. De versie die je nu in handen hebt werd geüpdatet in de zomer van 2024, na reeds een grote herziening in 2022.

Net zoals spreektaal, evolueert ook de programmeertaal C# constant. Terwijl ik dit schrijf zijn we aan versie 10.0 van C# en staat versie 11 in de startblokken. Bij iedere nieuwe C#-versie worden bepaalde concepten plots veel eenvoudiger of zelfs gewoon overbodig. Een goed programmeur moet natuurlijk zowel met de oude als de nieuwe constructies kunnen werken.

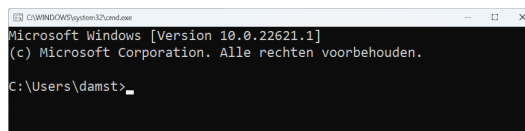
Ik heb getracht een gezonde mix tussen oud en nieuw te zoeken, waarbij de nadruk ligt op maximale bruikbaarheid in je verdere professionele carrière. Je zal hier dus geen stoere, state-of-the-art C# innovaties terugvinden die enkel in heel specifieke projecten bruikbaar zijn. Integendeel. Ik hoop dat als je aan het laatste hoofdstuk bent, je een zodanige basis hebt, dat je ook zonder problemen in andere 'zustertalen' durft te duiken (zoals Java, C en C++, maar ook zelfs Python of JavaScript).

Dit boek ambieert niet om de volledige C#-taal en alles dat daar rond hangt aan te leren. Het boek daarentegen is gericht op eender wie die interesse heeft in de wondere wereld van programmeren, maar mogelijk nog nooit één letter code effectief heeft geprogrammeerd. Bepaalde concepten die ik te ingewikkeld acht voor een beginnende programmeur werden dan ook weg gelaten. Beschouw wat je gaat lezen dus maar als een *gateway drug* naar meer C#, meer programmeertalen en vooral meer programmeerplezier! U weze gewaarschuwd.

1 Wat je kunt verwachten

Voor we verder gaan wil ik je wel even waarschuwen. Dit boek gaat uit van geen enkele kennis van programmeren, laat staan C#. Daarom beginnen we bij het prille begin. Verwacht echter niet dat je aan het einde van dit boek supercoole grafische applicaties of games kunt maken. Het is zelfs zo dat we hoegenaamd geen woord gaan reppen over “windows applicaties”, met knoppen en menu’s enz.

Alles dat in dit boek gemaakt wordt zal uitgevoerd “in de console”. Die oeroude DOS-schermen - ook wel een *shell* genoemd - die je nu nog vaak in films ziet wanneer hackers proberen in een erg beveiligd systeem in te breken. Deze aanpak helpt je te focussen op de essentie van het probleem, zonder afgeleid te worden door visuele elementen.



Figuur 1: De “console”. Qua zwarte inkt-verspilling zal deze afbeelding de hoofdprijs winnen!

2 Over de bronnen

Dit boek is het resultaat van bijna een decennium C# doceren aan de AP Hogeschool (eerst nog Hogeschool Antwerpen, dan Artesis Hogeschool, dan Artesis Plantijn Hogeschool, enz.). De eerste schrijfsels verschenen op een eigen gehoste blog (“Code van 1001 Nacht”, die ondertussen ter ziele is gegaan) en vervolgens kreeg deze een iets strakkere, eenduidige vorm als gitbook cursus.

Deze cursus, alsook een hele resem oefeningen en andere nuttige extra’s kan je terugvinden op **ziescherp.be**. De inhoud van die cursus loopt integraal gelijk aan die van dit boek. Uiteraard is de kans bestaande dat er in de online versie ondertussen weer wat minder schrijffoutjes staan.

Waarom deze korte historiek? Wel, de kans is bestaande dat er hier en daar flarden tekst, code voorbeelden, of oefeningen niet origineel de mijne zijn. Ik heb getracht zo goed mogelijk aan te geven wat van waar komt, maar als ik toch iets vergeten ben, aarzel dan niet om me er op te wijzen.

3 Benodigdheden

Alle codevoorbeelden in deze cursus kan je zelf (na)maken met de gratis **Visual Studio 2022 Community** editie die je kan downloaden op visualstudio.microsoft.com.

4 Dankwoord

Aardig wat mensen - grotendeels mijn eerstejaars studenten van de professionele bachelor Elektronica-ICT en Toegepaste Informatica van de AP Hogeschool - hebben me met deze cursus geholpen. Hen allemaal afzonderlijk bedanken zou me een extra pagina kosten, en ik heb de meeste al nadrukkelijk bedankt in de vorige editie van dit boek.

Een speciale dank nogmaals aan Maarten Wachters die de originele pixel-art van me maakte waar ik vervolgens enkele varianten op heb gemaakt.

Ook een bos bloemen voor collega's Olga Coutrin en Walter Van Hoof om de ondankbare taak op zich te nemen mijn vele dt-fouten uit de vorige editie te halen op nog geen week voor de deadline. Bedankt!

De trainers van Multimedi BV. die dit handboek ook gebruiken wil ik expliciet bedanken voor hun nuttige feedback op de eerste versie van dit boek, alsook om mij een extra reden te geven om dit boek in de eerste plaats uit te brengen.

Als laatste, in deze 2024 editie, een shoutout naar de leerkrachten van het middelbaar die sinds de laatste onderwijshervorming C# en OOP aan hun leerlingen mogen onderwijzen!



Veel lees-en programmeerplezier,

Tim Dams Zomer 2024

1 De eerste stappen

First, solve the problem. Then, write the code.

Wel, wel, wie we hier hebben?! Iemand die de edele kunst van het programmeren wil leren? Dan ben je op de juiste plaats gekomen. Je gelooft het misschien niet, maar reeds aan het einde van dit hoofdstuk zal je je eerste eigen computer-applicaties kunnen maken. De weg naar eeuwige roem, glorie, véél vloeken en code herbruiken ligt voor je. Ben je er klaar voor?

De eerste stappen zijn nooit eenvoudig. Ik probeer daarom het aantal dure woorden, vreemde afkortingen en ingewikkelde schema's tot een minimum te beperken. Maar toch. Als je een nieuwe kunst wil leren zal je handen én toetsenbord vuil moeten maken.

Wat er ook gebeurt de komende hoofdstukken: blijf volhouden. Leren programmeren is een beetje als een berg leren beklimmen waarvan je nooit de top lijkt te kunnen bereiken. Wat ook zo is. Er is geen “top”, en dat is net het mooie van dit alles. Er valt altijd iets nieuws te leren! De zaken waar je de komende pagina's op gaat vloeken zullen over enkele hoofdstukken al kinderspel lijken. Hou dus vol. Blijf oefenen. Vloek gerust af en toe. En vooral: geniet van het ontdekken van nieuwe dingen!

1.1 Wat is programmeren?

Je hoort de termen geregeld: softwareontwikkelaar, programmeur, app-developer, enz. Allen zijn beroepen die in essentie kunnen herleid worden tot hetzelfde: programmeren. Programmeurs hebben geleerd hoe ze computers opdrachten kunnen geven (**programmeren**) zodat deze hopelijk doen wat je ze vraagt.

In de 21e eeuw is de term *computer* natuurlijk erg breed. Quasi ieder apparaat dat op elektriciteit werkt bevat tegenwoordig een computertje. Gaande van slimme lampen, tot de servers die het Internet draaiende houden of de smartwatch aan je pols. Zelfs aardig wat ijskasten en wasmachines beginnen kleine computers te bevatten.

Het probleem van computers is dat het in essentie ongelooflijk domme dingen zijn. Hoe krachtig ze ook soms zijn. Ze zullen altijd **exact** doen wat jij hen vertelt dat ze moeten doen. Als je hen dus de opdracht geeft om te ontploffen, schrik dan niet dat je even later naar de 112 kunt bellen.

Programmeren houdt in dat je leert praten met die domme computers zodat ze doen wat jij wilt dat ze doen.

1.1.1 Het algoritme

Deze quote van John Johnson wordt door veel beginnende programmeurs soms met een scheef hoofd aanhoort. “Ik wil gewoon code schrijven!” Het is een mythe dat programmeurs constant code schrijven. Integendeel, een goed programmeur zal veel meer tijd in de “voorbereiding” tot code schrijven steken: het maken van een goed **algoritme** na een grondige **analyse van het probleem**.

Het algoritme is de essentie van een computerprogramma en kan je beschouwen als het recept dat je aan de computer gaat geven zodat deze jouw probleem op de juiste manier zal oplossen. **Het algoritme bestaat uit een reeks instructies** die de computer moet uitvoeren telkens jouw programma wordt uitgevoerd.

Het algoritme van een programma moet je zelf verzinnen. De volgorde waarin de instructies worden uitgevoerd zijn echter zeer belangrijk. Dit is exact hetzelfde als in het echte leven: een algoritme om je fiets op te pompen kan zijn:

- 1 Haal dop van het ventiel.
- 2 Plaats pomp op ventiel.
- 3 Begin te pompen.

Eender welke andere volgorde van bovenstaande algoritme zal vreemde - en soms fatale - fouten geven.

Wil je dus leren programmeren, dan zal je logisch moeten leren denken en een analytische geest hebben. Als je eerst tegen een bal trapt voor je kijkt waar de goal staat dan zal de edele kunst van het programmeren voor jou een...speciale aangelegenheid worden.¹

1.1.2 Programmeertaal

Om een algoritme te schrijven dat onze computer begrijpt dienen we een programmeertaal te gebruiken. Computers hebben hun eigen taaltje dat programmeurs moeten kennen voor ze hun algoritme aan de computer kunnen *voeden*. Er zijn tal van computertalen, de ene al wat obscuurder dan de andere. Maar wat al deze talen gelijk hebben is dat ze meestal:

- **ondubbelzinnig** zijn: iedere opdracht of woord kan door de computer maar op exact één manier geïnterpreteerd worden. Dit in tegenstelling tot bijvoorbeeld het Nederlands waar “wat een koele kikker” zowel een letterlijke, als een figuurlijke betekenis heeft die niets met elkaar te maken heeft.

¹Vanaf nu ben je trouwens gemachtigd om naar de nieuwsdiensten te mailen telkens ze foutief het woord “logaritme” gebruiken in plaats van “algoritme”. Het woord logaritme is iets wat bij sommige nachtmerries uit de lessen wiskunde opwekt en heeft hoegenaamd niets met programmeren te maken. Uiteraard kan het wel zijn dat je ooit een algoritme moet schrijven om een logaritme te berekenen. Hopelijk moet een journalist nooit voorgaande zin in een nieuwsbericht gebruiken.

- bestaan uit **woordenschat**: net zoals het Nederlands heeft ook iedere programmeertaal een lijst woorden die je kan gebruiken. Je gaat ook niet in het Nederlands zelf woorden verzinnen in de hoop dat je partner je kan begrijpen.
- bestaan uit **grammaticaregels**: Enkel Yoda mag Engels in een verkeerde volgorde gebruiken. Iedereen anders houdt zich best aan de grammatica-afspraken die een taal heeft. “bal rood is” lijkt nog begrijpbaar, maar als we zeggen “bal rood jongen is gooit veel”?

1.1.3 siesjarp

Net zoals er ontelbare spreektaalen in de wereld zijn, zijn er ook vele programmeertalen. **C#** - spreek uit ‘*siesjarp*’, soms ook *cs* geschreven - is er één van de vele. C# is een taal die deel uitmaakt van de .NET (spreek uit ‘*dotnet*’) . De .NET omgeving werd meer dan 20 jaar geleden door Microsoft ontwikkeld. Het fijne van C# is dat deze een zogenaamde **hogere programmeertaal** is. Hoe “hogere” de programmeertaal, hoe leesbaarder deze wordt voor leken omdat hogere programmeertalen dichter bij onze eigen taal aanleunen.

De geschiedenis van de hele .NET-wereld vertellen zou een boek op zich betekenen en gaan ik hier niet doen. Het is nuttig om weten dat er een gigantische bron aan informatie over .NET en C# online te vinden is².



Het fijne van leren programmeren is dat je binnenkort op een bepaald punt gaat komen waarbij de keuze van programmeertaal er minder toe doet. Vergelijk het met het leren van het Frans. Van zodra je Frans onder knie hebt is het veel eenvoudiger om vervolgens Italiaans of Spaans te leren. Zo ook met programmeertalen. De C# taal lijkt bijvoorbeeld als twee druppels water op Java. Ook de talen waar C# van afstamt - C en C++ - hebben erg herkenbare gelijkenissen.

Zelfs JavaScript, Python en veel andere moderne talen zullen weinig geheimen voor jou hebben wanneer je aan het einde van dit boek bent.

1.1.4 Anders Hejlsberg

Deze Deen krijgt een eigen sectie in dit boek. Waarom? Hij is niemand minder dan de “uitvinder” van C#. Anders Hejlsberg heeft een stevig palmares inzake programmeertalen verzinnen. Voor hij C# boven het doopvont hield bij Microsoft, schreef hij ook al Turbo Pascal én was hij de *chief architect* van Delphi.

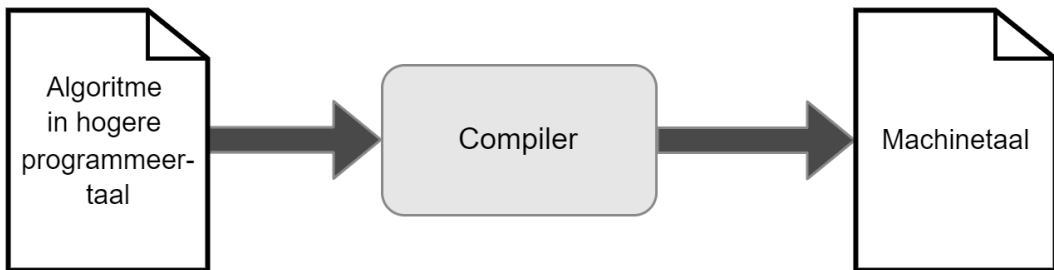
Je zou denken dat hij na 3 programmeertalen wel op z’n lauweren zou rusten, maar zo werkt Anders niet. In 2012 begon hij te werken aan een JavaScript alternatief, wat uiteindelijk het immens populaire TypeScript werd. Dit allemaal om maar te zeggen dat als je één poster in je slaapkamer moet ophangen, het die van Anders zou moeten zijn.

²Zie docs.microsoft.com/en-us/dotnet/csharp/getting-started.

1.1.5 De compiler

Rechtstreeks onze algoritmen tegen de computer vertellen vereist dat we machinetaal kunnen. Deze is echter zo complex dat we tientallen lijnen machinetaal nodig hebben om nog maar gewoon 1 letter op het scherm te krijgen. Er werden daarom dus hogere programmeertalen ontwikkeld die aangenamer zijn dan deze zogenaamde machinetalen om met computers te praten.

Uiteraard hebben we een vertaler nodig die onze code zal vertalen naar de machinetaal van het apparaat waarop ons programma moet draaien. Deze vertaler is de **compiler** die aardig wat complex werk op zich neemt, maar dus in essentie onze code gebruiksklaar maakt voor de computer.



Figuur 1.1: Vereenvoudigd compiler overzicht.

Merk op dat ik hier veel details van de compiler achterwege laat. De compiler is een uitermate complex element. In deze fase van je programmeursleven hoeven we enkel de kern van de compiler te begrijpen: **het omzetten van C# code naar een uitvoerbaar bestand geschreven in machinetaal.**



Microsoft .NET

Bij de geboorte van .NET in 2000 kwam ook de taal C#.

.NET is een **framework** dat bestaat uit een grote verzameling bibliotheken (*class libraries*) en een *virtual execution system* genaamd de **Common Language Runtime (CLR)**. De CLR zal ervoor zorgen dat C# en .NET talen (bv. F# en Visual Basic.NET) kunnen samenwerken met de vele bibliotheken.

Om een uitvoerbaar bestand te maken (**executable**) zal de broncode die je hebt geschreven in C# worden omgezet naar **Intermediate Language (IL)** code. Op zich is deze IL code nog niet uitvoerbaar, maar dat is niet ons probleem.

Wanneer een gebruiker een in IL geschreven bestand wil uitvoeren dan zal de CLR achter de schermen deze code ogenblikkelijk naar machine code omzetten en uitvoeren. Dit concept noemt men **Just-In-Time** of JIT compilatie. De gebruiker zal dus nooit dit proces opmerken (tenzij er geen .NET framework werd geïnstalleerd op het systeem).

1.1.6 Nummering en naamgeving van C#

Microsoft heeft er een handje van weg om hun producten ingewikkelde volgnummers-of letters te geven, denk maar aan Windows 10 die de opvolger was van Windows 8 (dat had trouwens een erg goede reden; zoek maar eens op), of Windows 7 dat Windows Vista opvolgde. Het helpt ook niet dat ze geregeld hun producten een nieuwe naam geven. Zo was het binnen .NET tot voor kort erg ingewikkeld om te weten welke versie nu eigenlijk de welke was.

Microsoft heeft gelukkig recent de naamgevingen herschikt én hernoemt in de hoop het allemaal wat duidelijker te maken. Ik zal daarom even kort te bespreken waar we nu zitten.

.NET 6 (framework)

Telkens er een nieuwe .NET framework werd *gereleased* verscheen er ook een bijhorende nieuwe versie van Visual Studio. Vroeger had je verschillende frameworks binnen de .NET familie zoals *.NET Framework*, *“.NET Standard”*, *.NET Core* enz. die allemaal net niet dezelfde doeleinden hadden wat het erg verwarrend maakte. Om dit te vereenvoudigen bestaat sinds 2020 enkel nog .NET gevolgd door een nummer.

Zo had je in 2020 .NET 5 en verschijnt eind 2022 .NET 7. Dit boek maakt gebruik van **.NET 6** dat verscheen samen met Visual Studio 2022...in november 2021. Je moet er maar aan uit kunnen.

C# 10

De C# taal is eigenlijk nog het eenvoudigst qua nummering. Om de zoveel tijd krijgt C# een update met een nieuwe reeks taal-eigenschappen die je kan, maar niet hoeft te gebruiken. Momenteel zitten we aan **C# 10** dat werd uitgebracht samen met .NET 6.

Eind 2023 kwam .NET 8 uit en dus ook alweer een nieuwe versie van C#, namelijk versie 12. De kans is dus groot dat voorgaande zin alweer gedateerd is tegen dat je hem leest. De vernieuwingen in C# zijn niet altijd belangrijk voor beginnende programmeurs. In dit boek heb ik getracht de belangrijkste én meest begrijpbare nieuwe features uit de taal te gebruiken waar relevant. Over het algemeen gezien mag je stellen dat dit boek tot en met versie .NET 7.3 / C# versie 11 de belangrijkste zaken zal behandelen.



Je vraagt je misschien af waarom dit allemaal verteld wordt? Waarom wordt deze geschiedenisles gegeven? De reden is heel eenvoudig. Je gaat zeker geregeld zaken op het internet willen opzoeken tijdens het (leren) programmeren en zal dan ook vaker op artikels stuiten met de oude(re) naamgeving en dan mogelijks niet kunnen volgen.

1.2 Kennismaken met C# en Visual Studio

Je gaat in dit boek leren programmeren met Microsoft Visual Studio 2022, een softwarepakket waar ook een gratis community versie voor bestaat. Microsoft Visual Studio (vanaf nu **VS**) is een pakket dat een groot deel van de tools samenvoegt die een programmeur nodig heeft. Zo zit er een onder andere een debugger, code editor en compiler in.

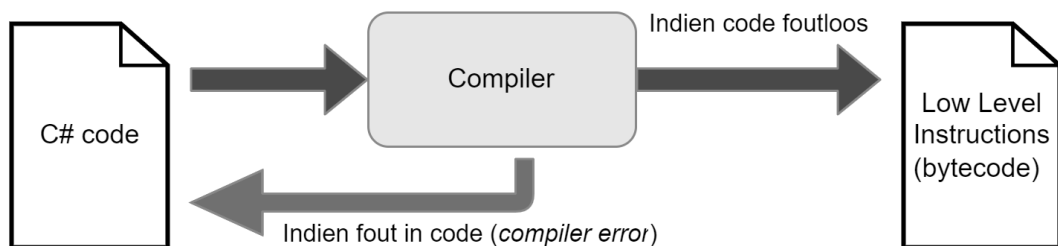
VS is een zogenaamde **IDE** ("**I**ntegrated **D**evelopment **E**nvironment") en is op maat gemaakt om in C# geschreven applicaties te ontwikkelen. Je bent echter verre van verplicht om enkel C# applicaties in VS te ontwikkelen. Je kan gerust VB.NET, TypeScript, Python en andere talen gebruiken. Ook vice versa ben je niet verplicht om VS te gebruiken om te ontwikkelen. Je kan zelfs in notepad code schrijven en vervolgens compileren. Er bestaan zelfs online C# programmeer omgevingen, zoals **dotnetfiddle.net**.

1.2.1 De compiler en Visual Studio

Zoals gezegd: jouw taak als programmeur is algoritmes in C# taal uitschrijven. Je zou dit in een eenvoudige tekstverwerker kunnen doen, maar dan maak je het jezelf lastig. Net zoals je tekst in notepad kunt schrijven, is het handiger dit bijvoorbeeld in tekstverwerker zoals Word te doen: je krijgt een spellingchecker en allerlei handige extra's.

Ook voor het schrijven van computer code is het handiger om een IDE te gebruiken, een omgeving die ons zal helpen foutloze C# code te schrijven.

Het hart van Visual Studio bestaat uit de compiler die ik hiervoor besprak. De compiler zal je C# code omzetten naar de IL-code zodat je je applicatie op een computer kunnen gebruiken. Zolang je C# code niet exact voldoet aan de C# syntax en grammatica zal de compiler het vertikken een uitvoerbaar bestand voor je te genereren.

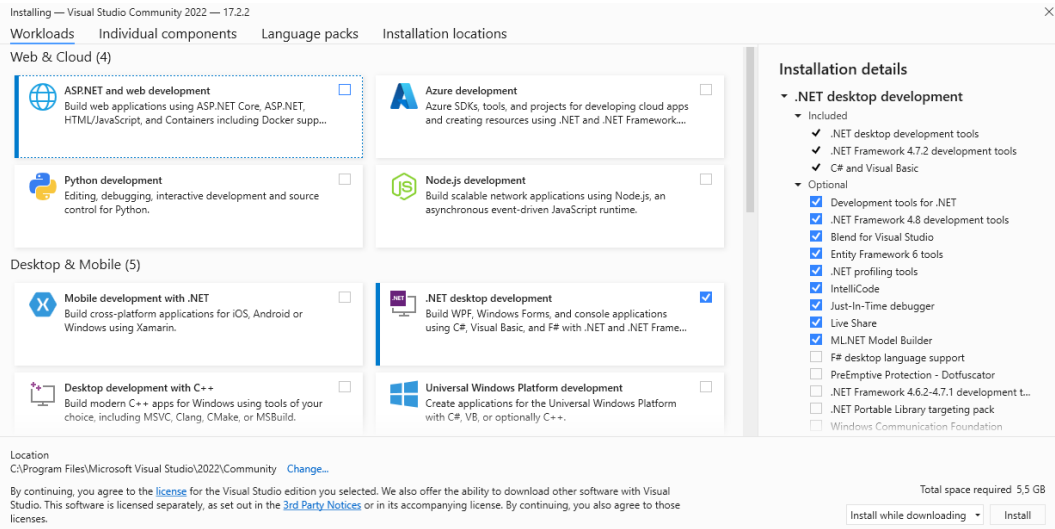


Figuur 1.2: Vereenvoudigd compiler overzicht.

1.2.2 Visual Studio Installeren

In dit boek zullen de voorbeelden steeds met de **Community** editie van VS gemaakt zijn. Je kan deze gratis downloaden en installeren via visualstudio.microsoft.com/vs.

Het is belangrijk bij de installatie dat je zeker de **.NET desktop development** workload kiest. Uiteraard ben je vrij om meerdere zaken te installeren.



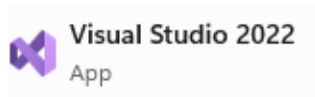
Figuur 1.3: In dit boek zullen we enkel met de .NET desktop development workload werken.



In dit boek zullen we dus steeds werken met *Visual Studio Community 2022*. Niet met **Visual Studio Code**. Visual Studio code is een zogenaamde lightweight versie van VS die echter zeker ook z'n voordelen heeft. Zo is VS Code makkelijk uitbreidbaar, snel, en compact. Visual Studio vindt dankzij VS Code eindelijk ook z'n weg op andere platformen dan enkel die van Microsoft. Je kan de laatste versie ervan downloaden op: code.visualstudio.com.

1.2.3 Visual studio opstarten

Als alles goed is geïnstalleerd kan je Visual Studio starten via het start-menu van Windows.



Figuur 1.4: “We are going on an adventure!” (Bron: Bilbo Baggins)

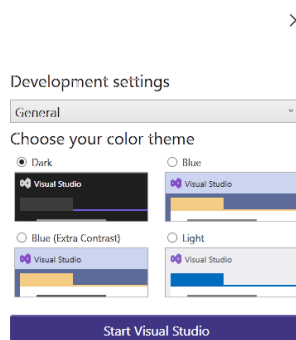
De allereerste keer dat je VS opstart krijg je 2 extra schermen te zien:

- Het “sign in” scherm mag je overslaan. Kies “Not now, maybe later”.
- Op het volgende scherm kies je best als “Development settings” voor **Visual C#**. Vervolgens kan je je kleurenthema kiezen. Dit heeft geen invloed op de manier van werken.



Dark is uiteraard het coolste thema om in te coderen. Je voelt je ogenblikkelijk Neo uit The Matrix. Het nadeel van dit thema is dat het veel meer inkt verbruikt indien je screenshots in een boek zoals dit wilt plaatsen.

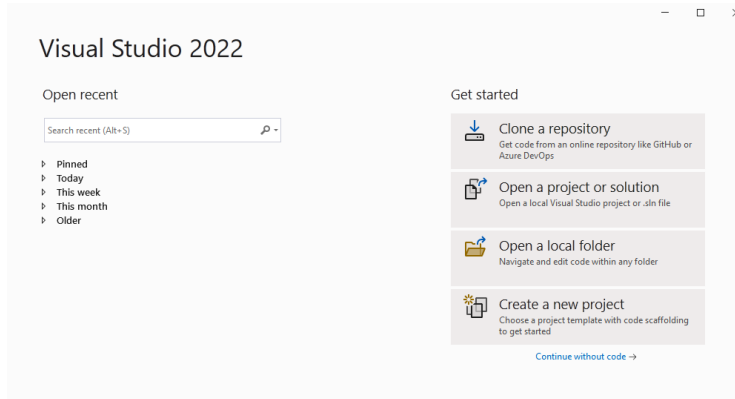
De keuze voor Development Setting kan je naar “Visual C#” veranderen, maar General is even goed (je zal geen verschil merken in eerste instantie). Je kan dit achteraf nog aanpassen in VS via “Tools” in de menubalk, dan “Import and Export Settings” en kiezen voor “Import and Export Settings Wizard”.



Figuur 1.5: Je kan dit nadien ook altijd nog aanpassen. En zelfs personaliseren tot de vreemdste kleur- en lettertypecombinaties.

1.2.3.1 Project keuze

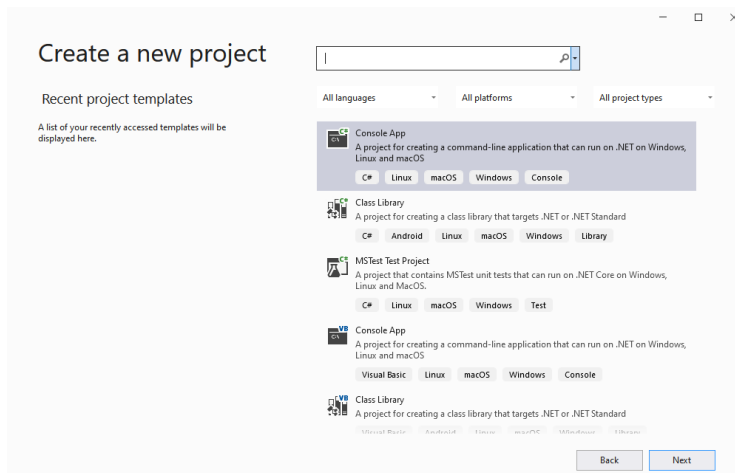
Na het opstarten van VS krijg je het startvenster te zien van waaruit je verschillende dingen kan doen. Van zodra je projecten gaat aanmaken zullen deze in de toekomst ook op dit scherm getoond worden zodat je snel naar een voorgaand project kunt gaan.



Figuur 1.6: Het startscherm van Visual Studio.

1.2.3.2 Een nieuw project aanmaken

We zullen nu een nieuw project aanmaken, kies hiervoor “Create a new project”.

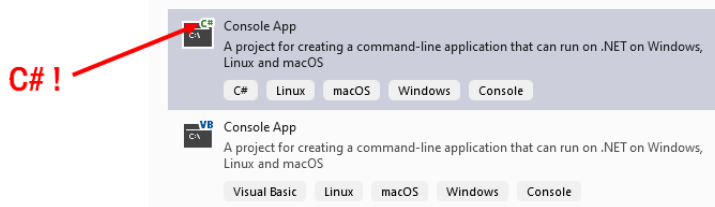


Figuur 1.7: Kies je projecttype.



Het “New Project” venster dat nu verschijnt geeft je hopelijk al een glimp van de veelzijdigheid van VS. In het rechterdeel zie je bijvoorbeeld alle Project Types staan. M.a.w. dit zijn alle soorten programma’s die je kan maken in VS. Naargelang de geïnstalleerde opties en bibliotheken zal deze lijst groter of kleiner zijn.

In dit boek zal je altijd het Project Type “**Console App**” gebruiken (ZONDER .NET Framework achteraan). Je vindt deze normaal bovenaan de lijst terug, maar kunt deze ook via het zoekveld bovenaan terugvinden. Zoek gewoon naa - je raadt het nooit - *console*. **Let er op dat je een klein groen C# icoontje ziet staan bij het zwarte icoon van de Console app.** Ook andere talen ondersteunen console applicaties, maar wij gaan natuurlijk met C# aan het werk.



Figuur 1.8: Kies voor C#, niet Visual Basic (VB). Dank bij voorbaat!

Een console applicatie is een programma dat alle uitvoer naar een zogenaamde *console* stuurt, een shell. Je kan met andere woorden enkel tekst als uitvoer genereren. Multimedia elementen zoals afbeeldingen, geluid en video zijn dus uit den boze.

Kies dit type en klik ‘Next’.

Op het volgende scherm kan je een naam ingeven voor je project alsook de locatie op de harde schijf waar het project dient opgeslagen te worden. **Onthoud waar je je project aanmaakt zodat je dit later terugvindt.**

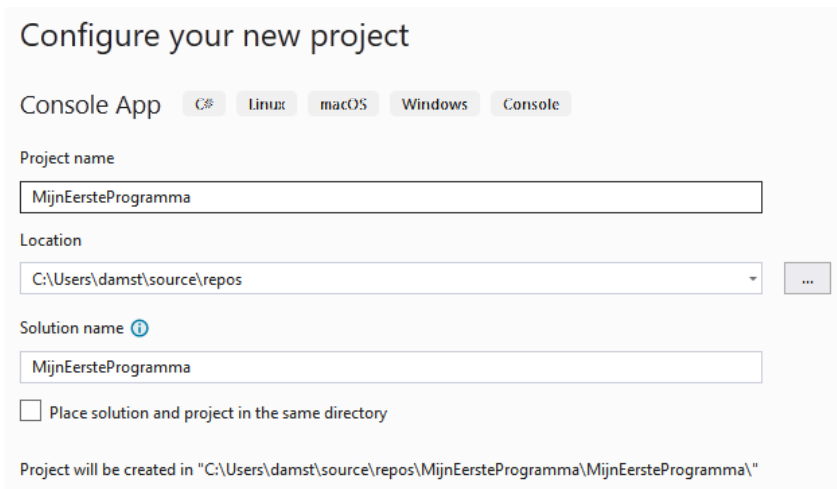


Het “Solution name” tekstveld blijf je af. Hier zal automatisch dezelfde tekst komen als die dat je in het “Project name” tekstveld invult.



Geef je projectnamen ogenblikkelijk duidelijke namen zodat je niet opgezadeld geraakt met projecten zoals Project201, enz. waarvan je niet meer weet welke belangrijk zijn en welke niet.

Geef je project de naam “MijnEersteProgramma” en kies een goede locatie. **Ik raad aan om de checkbox “Place solution and project in the same directory” onderaan niét aan te vinken.** In de toekomst zal het nuttig zijn dat je meer dan 1 project per solution zal kunnen hebben. Lig er nog niet van wakker.



Configure your new project

Console App C# Linux macOS Windows Console

Project name
MijnEersteProgramma

Location
C:\Users\damst\source\repos

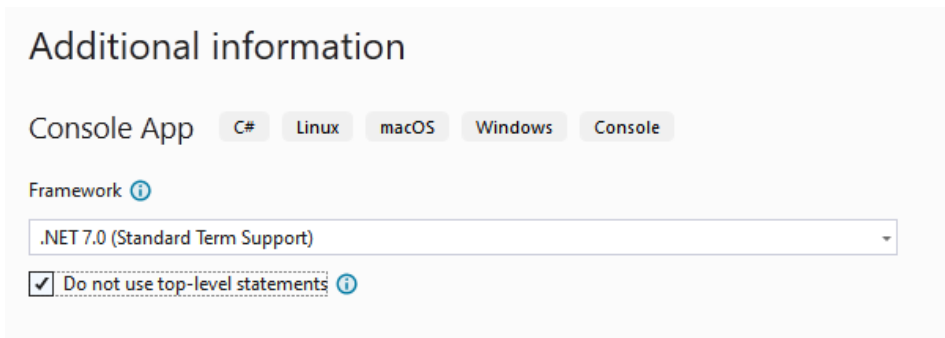
Solution name ⓘ
MijnEersteProgramma

☐ Place solution and project in the same directory

Project will be created in "C:\Users\damst\source\repos\MijnEersteProgramma\MijnEersteProgramma"

Figuur 1.9: Kijk altijd goed na waar je je solution gaat plaatsen.

Klik op next en kies als Target Framework de meest recente versie. **Duidt hier zeker de checkbox aan met “Do not use top level statements”!!!³**. Klik nu op Create.



Additional information

Console App C# Linux macOS Windows Console

Framework ⓘ
.NET 7.0 (Standard Term Support)

☒ Do not use top-level statements ⓘ

Figuur 1.10: Gebruik alsjeblieft geen top-level statements!

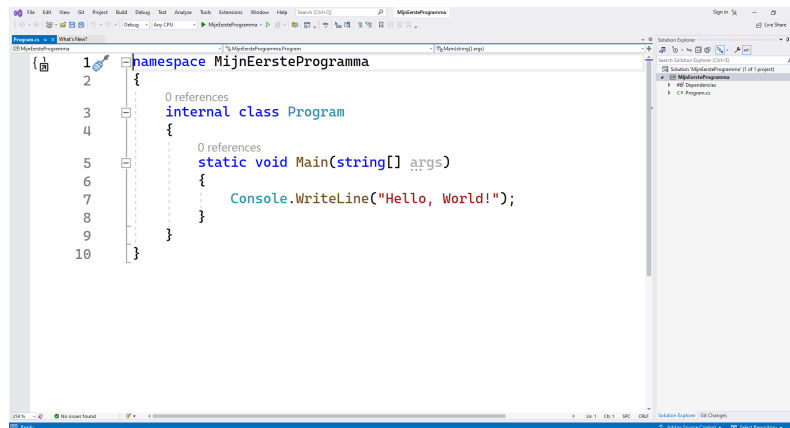
VS heeft nu reeds een aantal bestanden aangemaakt die je nodig hebt om een ‘Console Applicatie’ te maken.

³De auteur van dit boek kan fier melden dat die checkbox er staat mede dankzij zijn gezaag op github.com/dotnet/docs/issues/2742.

1.2.4 IDE Layout

Wanneer je VS opstart zal je mogelijk overweldigd worden door de hoeveelheid menu's, knopjes, schermen, enz. Dit is normaal voor een IDE: deze wil zoveel mogelijk mogelijkheden aanbieden aan de gebruiker. Vergelijk dit met Word: afhankelijk van wat je gaat doen gebruikt iedere gebruiker andere zaken van Word. De makers van Word kunnen dus niet bepaalde zaken weglaten, ze moeten net zoveel mogelijk aanbieden.

Eens kijken wat we allemaal zien in VS na het aanmaken van een nieuw programma...



Figuur 1.11: VS IDE overzicht.

- Je kan meerdere bestanden tegelijkertijd openen in VS. Ieder bestand zal z'n eigen **tab** krijgen. De actieve tab is het bestand wiens inhoud je in het hoofdgedeelte eronder te zien krijgt. Merk op dat enkel open bestanden een tab krijgen. Je kan deze tabbladen ook "lostrekken" om bijvoorbeeld enkel dat tabblad op een ander scherm te plaatsen.
- De "**solution explorer**" aan de rechterzijde toont alle bestanden en elementen die tot het huidige project behoren. Als we dus later nieuwe bestanden toevoegen, dan kan je die hier zien en openen. Verwijder hier géén bestanden zonder dat je zeker weet wat je aan het doen bent!



Indien je een nieuw project hebt aangemaakt en de code die je te zien krijgt lijkt in de verste verte niet op de code die je hierboven ziet dan heb je vermoedelijk een verkeerd projecttype of taal gekozen. Of je hebt de "Do not use top-level statements" checkbox niet aangeduid.



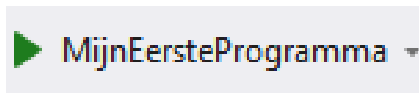
Layout kapot/kwijt/vreemd?

De layout van VS kan je volledig naar je hand zetten. Je kan ieder (deel-)venster en tab verzetten, verankeren en zelfs verplaatsen naar een ander bureaublad. Experimenteer hier gerust mee en besef dat je steeds alles kan herstellen. Het gebeurt namelijk al eens dat je layout een beetje om zeep is:

- Om eenvoudig een venster terug te krijgen, bijvoorbeeld het properties window of de solution explorer: klik bovenaan in de menubalk op “View” en kies dan het gewenste venster (soms staat dit in een submenu).
- Je kan ook altijd je layout in z’n geheel **resetten**: ga naar “Window” en kies “Reset window layout”.

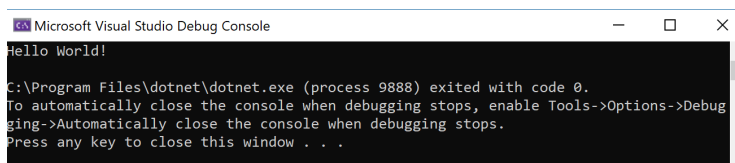
1.2.5 Je programma starten

De code in Program.cs die VS voor je heeft gemaakt is reeds een werkend programma. Erg nuttig is het helaas nog niet. Je kan de code compileren en uitvoeren door op de groene driehoek bovenaan te klikken:



Figuur 1.12: Het programma uitvoeren.

Als alles goed gaat krijg je nu “Hello World!” te zien en wat extra informatie omtrent het programma dat net werd uitgevoerd:



Figuur 1.13: Uitvoer van het programma.

Veel doet je programma nog niet natuurlijk, dus sluit dit venster maar terug af door een willekeurige toets in te drukken.

1.2.6 Is dit alles?

Nee hoor. Visual Studio is lekker groot, maar laat je dat niet afschrikken. Net zoals voor het eerst op een nieuwe reisbestemming komen, kan deze in het begin overweldigend zijn. Tot je weet waar het zwembad en de pingpongtafel staat en je van daaruit begint te oriënteren.

1.3 Console-applicaties

Een console-applicatie is een programma dat zijn in- en uitvoer via een klassiek commando/shell-scherm toont. Zoals al verteld: in dat boek ga ik je enkel console-applicaties leren maken. Grafische Windows applicaties komen niet aan bod.

1.3.1 In en uit - ReadLine en WriteLine

Een programma zonder invoer van de gebruiker is niet erg boeiend. De meeste programma's die we leren schrijven vereisen dan ook "input" (**IN**). We moeten echter ook zaken aan de gebruiker kunnen tonen. Denk bijvoorbeeld aan een foutboodschap of de uitkomst van een berekening tonen. Dit vereist dat er ook "output" (**UIT**) naar het scherm kan gestuurd worden.



Figuur 1.14: In het begin zullen al je applicaties deze opbouw hebben.

Console-applicaties maken in C# vereist dat je minstens twee belangrijke C# methoden leert gebruiken:

- Met behulp van **Console.ReadLine()** kunnen we input van de gebruiker inlezen en in ons programma verwerken.
- Via **Console.WriteLine()** kunnen we tekst op het scherm tonen.

1.3.2 Je eerste console programma

Sluit het eerder gemaakte “MyFirstProject” project af en herstart Visual Studio. Maak nu een nieuw console-project aan. Noem dit project *Demo1*. Open het Program.cs bestand via de solution Explorer (indien het nog niet open is). **Veeg de code die hier reeds staat niet weg!**

Voeg onder de lijn `Console.WriteLine("Hello World!");` volgende code toe (vergeet de puntkomma niet):

```
1 Console.WriteLine("Hoi, ik ben het!");
```

Zodat je dus volgende code krijgt:

```
1 namespace Demo1
2 {
3     internal class Program
4     {
5         static void Main(string[] args)
6         {
7             Console.WriteLine("Hello World!");
8             Console.WriteLine("Hoi, ik ben het");
9         }
10    }
11 }
```

Compileer deze code en voer ze uit: **druk hiervoor weer op het groene driehoekje bovenaan.** Of via het menu Debug en dan Start Debugging.



Moet ik niets bewaren?

Neen. Telkens je op de groene “build en run” knop duwt worden al je aanpassingen automatisch bewaard. Trouwens: **Kies nooit voor “save as...”!** want dan bestaat de kans dat je project niet meer compileert. Dit zal aardig wat problemen in je project voorkomen, geloof me maar.



Laat je niet afschrikken door wat er nu volgt. Ik gooi je even in het diepe gedeelte van het zwembad maar zal je er op tijd uithalen. Vervolgens kunnen we terug in het babybadje rustig op de glijbaan kunnen gaan spelen en C# op een trager tempo verder ontdekken.

1.3.2.1 Analyse van de code

Ik zal nu iedere lijn code kort bespreken. Sommige lijnen code zullen lange tijd niet belangrijk zijn. Onthoud nu alvast dat: **alle belangrijke code staat tussen de accolades onder de lijn `static void Main(string[] args)`!**

- **Lijn 1:** Dit is de unieke naam waarbinnen we ons programma zullen plaatsen, en het is niet toevallig de naam van je project. Verander dit nooit tenzij je weet wat je aan het doen bent. Ik bespreek *namespaces* in hoofdstuk 10.
- **Lijn 3:** Hier start je echte programma. Alle code binnen deze Program accolades zullen gecompileerd worden naar een uitvoerbaar bestand. Vanaf hoofdstuk 9 zal deze lijn geen geheimen meer hebben voor je.
- **Lijn 5:** Het startpunt van iedere console-applicatie. Wat hier gemaakt wordt is een **methode** genaamd `Main`. Je programma kan meerdere methoden (of functies) bevatten, maar enkel degene genaamd `Main` zal door de compiler als het startpunt van het programma gemaakt worden. Deze lijn zal ik in hoofdstuk 7 en hoofdstuk 8 uit de doeken doen.
- **Lijn 7:** Dit is een **statement** dat de `WriteLine`-methode aanroept van de `Console`-bibliotheek. Het zal alle tekst die tussen de aanhalingstekens staat op het scherm tonen.
- **Lijn 8:** en ook deze lijn zorgt ervoor dat er tekst op het scherm komt wanneer het programma zal uitgevoerd worden.
- **Accolades** op lijnen 2, 4, 6, 9 tot en met 10: vervolgens moet voor iedere openende accolade eerder in de code nu ook een bijhorende sluitende volgen. We gebruiken accolades om de *scope* aan te duiden, iets dat we in hoofdstuk 5 geregeld zullen nodig hebben.

Net zoals een recept, zal ook in C# code van **boven naar onder worden uitgevoerd**.

Voor ons wordt het echter pas interessant op lijn 7⁴. Dit is het startpunt van ons programma en de uitvoer ervan. Al de zaken ervoor kan je voorlopig keihard negeren.

Het programma zal alles uitvoeren dat tussen de accolades van het `Main`-blok staat. Dit blok wordt afgebakend door de accolades van lijn 6 en 9. Dit wil ook zeggen dat van zodra lijn 9 wordt bereikt, dit het signaal voor je computer is om het programma af te sluiten.

⁴“Hello world” op het scherm laten verschijnen wanneer je een nieuwe programmeertaal leert is ondertussen een traditie bij programmeurs. Er is zelfs een website die dit verzamelt namelijk **helloworldcollection.de**. Deze site toont in honderden programmeertalen hoe je “Hello world” moet programmeren.



Jawadde... Wat was dit allemaal?! We hebben al aardig wat vreemde code zien passeren en het is niet meer dan normaal dat je nu denkt "dit ga ik nooit kunnen". Wees echter niet bevreesd: je zal sneller dan je denkt bovenstaande code als 'kinderspel' gaan bekijken. Een tip nodig? Test en experimenteer met wat je al kunt!

Laat deze info rustig inzinken en onthoud alvast volgende belangrijke zaken:

- ***Al je eigen code komt momenteel enkel tussen de `Main` accolades.***
- ***Eindig iedere lijn code daar met een puntkomma (;).***
- ***Code wordt van boven naar onder uitgevoerd.***



De oerman verschijnt wanneer we een stevige stap gezet hebben en je mogelijk even onder de indruk bent van al die nieuwe informatie. Hij zal proberen informatie nog eens vanuit een ander standpunt toe te lichten en te herhalen waarom deze nieuwe kennis zo belangrijk is.

1.3.3 WriteLine: Tekst op het scherm

De `WriteLine`-methode is een veelgebruikte methode in Console-applicaties. Het zorgt ervoor dat we tekst op het scherm kunnen tonen.

Voeg volgende lijn toe na de vorige `WriteLine`-lijn in je project:

```
1 Console.WriteLine("Wie ben jij?!");
```

De `WriteLine` methode zal alle tekst tonen die tussen de aanhalingstekens (" ") staat. **De aanhalingstekens aan het begin en einde van de tekst zijn uiterst belangrijk! Alsook het puntkomma helemaal achteraan.**

Je code binnen de `Main` accolades zou nu moeten zijn:

```
1 Console.WriteLine("Hello World!");  
2 Console.WriteLine("Hoi, ik ben het");  
3 Console.WriteLine("Wie ben jij?!");
```

Kan je voorspellen wat de uitvoer zal zijn? Test het eens!



Ik toon niet telkens de volledige broncode. Als ik dat zou blijven doen dan wordt dit boek dubbel zo dik. Ik toon daarom (meestal) enkel de code die binnen de `Main` (of later ook elders) moet komen.

1.3.4 ReadLine: Input van de gebruiker verwerken

In de Console kan je met een handvol methoden reeds een aantal interessante dingen doen.

Zo kan je bijvoorbeeld input van de gebruiker inlezen en bewaren in een variabele als volgt:

```
1 string result;  
2 result = Console.ReadLine();
```

Wat gebeurt er hier juist?

De eerste lijn code:

- Concreet zeggen we hiermee aan de compiler: maak in het geheugen een plekje vrij waar enkel data van het type `string` in mag bewaard worden (wat deze zin exact betekent komt later. Onthoud nu dat geheugen van het type `string` enkel “tekst” kan bevatten).
- Noem deze geheugenplek `result` zodat we deze later makkelijk kunnen in en uitlezen.

Tweede lijn code:

- Vervolgens roepen we de `ReadLine` methode aan. Deze methode zal de invoer van de gebruiker van het toetsenbord uitlezen tot de gebruiker op enter drukt.
- Het resultaat van de ingevoerde tekst wordt bewaard in de variabele `result`.



Merk op dat de toekenning in C# van rechts naar links gebeurt. Vandaar dat `result` dus links van de toekenning (`=`) staat en de waarde krijgt van het gedeelte rechts ervan.

Je programma zou nu moeten zijn:

```
1 Console.WriteLine("Hello World!");  
2 Console.WriteLine("Hoi, ik ben het!");  
3 Console.WriteLine("Wie ben jij?!");  
4 string result;  
5 result = Console.ReadLine();
```

Start nogmaals je programma. Je zal merken dat je programma nu een cursor toont en wacht op invoer nadat het de eerste 3 lijnen tekst op het scherm heeft gezet. Je kan nu eender wat intypen en van zodra je op enter duwt gaat het programma verder. Maar aangezien lijn 5 de laatste lijn van ons algoritme is, zal je programma hierna afsluiten. We hebben dus de gebruiker voor niets iets laten invoeren.

1.3.5 Input gebruiker gebruiken

Een variabele is een geheugenplekje met een naam waar we zaken in kunnen bewaren. In het volgende hoofdstuk gaan we zo vaak het woord variabele gebruiken dat je oren en ogen er van gaan bloeden. Trek je nu dus nog niet te veel aan van dit woord. We kunnen nu invoer van de gebruiker gebruiken en tonen op het scherm. De invoer hebben we bewaard in de variabele `result`:

```
1 Console.WriteLine("Dag");  
2 Console.WriteLine(result);  
3 Console.WriteLine("hoe gaat het met je?");
```

In de tweede lijn hier gebruiken we de variabele `result` als parameter in de `WriteLine`-methode.

Met andere woorden: de `WriteLine` methode zal op het scherm tonen wat de gebruiker even daarvoor heeft ingevoerd.

Je volledige programma ziet er dus nu zo uit:

```
1 Console.WriteLine("Hello World!");  
2 Console.WriteLine("Hoi, ik ben het!");  
3 Console.WriteLine("Wie ben jij?!");  
4 string result;  
5 result = Console.ReadLine();  
6 Console.WriteLine("Dag ");  
7 Console.WriteLine(result);  
8 Console.WriteLine("hoe gaat het met je?");
```

Test het programma en voer je naam in wanneer de cursor knippert.

Voorbeelduitvoer (lijn 3 is wat de gebruiker heeft ingetypd)

```
1 Hoi, ik ben het!  
2 Wie ben jij?!  
3 tim [enter]  
4 Dag  
5 tim  
6 hoe gaat het met je?
```

1.3.6 Aanhalingsteken of niet?

Wanneer je de inhoud van een variabele wil gebruiken in een methode zoals `WriteLine()` dan plaats je deze zonder aanhalingstekens!

Bekijk zelf eens wat het verschil wordt wanneer je volgende lijn code `Console.WriteLine(result);` vervangt door `Console.WriteLine("result");`.

De uitvoer wordt dan:

```
1 Hoi, ik ben het!  
2 Wie ben jij?!  
3 tim [enter]  
4 Dag  
5 result  
6 hoe gaat het met je?
```

We krijgen dus letterlijk de tekst “result” op het scherm in plaats van de gebruikersinvoer die we in de variabele bewaarden.

1.3.7 Write en WriteLine

Naast `WriteLine` bestaat er ook `Write`.

De `WriteLine`-methode zal steeds een *line break* - een *enter* zeg maar - aan het einde van de lijn zetten zodat de cursor naar de volgende lijn springt.

De `Write`-methode daarentegen zal geen enter aan het einde van de lijn toevoegen. Als je dus vervolgens iets toevoegt met een volgende `Write` of `WriteLine`, **dan zal dit aan dezelfde lijn toegevoegd worden.**

Vervang daarom eens in de laatste 3 lijnen code in je project `WriteLine` door `Write`:

```
1 Console.WriteLine("Dag");  
2 Console.WriteLine(result);  
3 Console.WriteLine("hoe gaat het met je?");
```

Voer je programma uit en test het resultaat. Je krijgt nu:

```
1 Hoi, ik ben het!  
2 Wie ben jij?!  
3 tim [enter]  
4 Dagtimhoe gaat het met je?
```

Wat is er hier “verkeerd” gelopen? Al je tekst van de laatste lijn plakt zo dicht bij elkaar?

Inderdaad, ik ben spaties vergeten toe te voegen. Spaties zijn ook tekens die op scherm moeten komen - ook al zien we ze niet - en je dient dus binnen de aanhalingstekens spaties toe te voegen.

Namelijk:

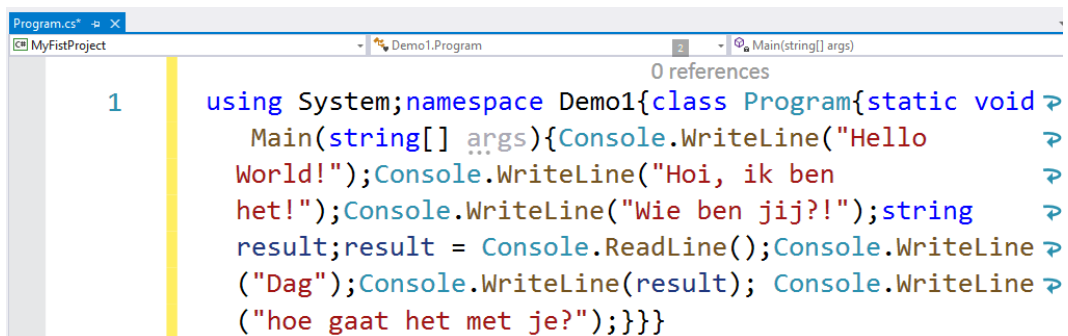
```
1 Console.Write("Dag ");
2 Console.Write(result);
3 Console.Write(" hoe gaat het met je?");
```

Je uitvoer wordt nu:

```
1 Hoi, ik ben het!
2 Wie ben jij?!
3 tim [enter]
4 Dag tim hoe gaat het met je?
```

1.3.8 Witregels in C#

C# trekt zich niets aan van **witregels die níét binnen aanhalingstekens staan**. Zowel spaties, enters en tabs worden genegeerd. Met andere woorden: je kan het voorgaande programma perfect in één lange lijn code typen, zonder enters. Dit is echter niet aangeraden want het maakt je code een pak onleesbaarder.



Figuur 1.15: Voorgaande programma in exact 1 lijn. Cool? Ja, in sommige kringen. Dom en onleesbaar? Ook ja.



Opletten met spaties

Let goed op hoe je spaties gebruikt bij `WriteLine`. **Indien je spaties buiten de aanhalingstekens plaatst dan heeft dit geen effect.**

Hier een fout gebruik van spaties (de code zal werken maar je spaties worden genegeerd):

```
1 //we visualiseren de spaties even als liggende streepjes
  in volgende voorbeeld
2 Console.Write("Dag_");
3 Console.Write(result_);
4 Console.Write("hoe gaat het met je?");
```

En een correct gebruik:

```
1 Console.Write("Dag_");
2 Console.Write(result);
3 Console.Write("_hoe gaat het met je?");
```

1.3.9 Zinnen aan elkaar plakken

We kunnen dit allemaal nog een pak korter tonen zonder dat de code onleesbaar wordt. De plus-operator (+) in C# kan je namelijk gebruiken om tekst achter elkaar te *plakken*. De laatste 3 lijnen code kunnen dan korter geschreven worden als volgt:

```
1 Console.WriteLine("Dag " + result + " hoe gaat het met je?");
```

Merk op dat `result` dus NIET tussen aanhalingstekens staat, in tegenstelling tot de andere stukken van de zin. Waarom is dit? Aanhalingstekens in C# duiden aan dat een stuk tekst moet beschouwd worden als tekst van het type **string**. Als je geen aanhalingsteken gebruikt dan zal C# de tekst beschouwen als een variabele met die naam.

Bekijk zelf eens wat het verschil wordt wanneer je volgende lijn code:

```
1 Console.WriteLine("Dag "+ result + " hoe gaat het met je?");
```

Vervangt door:

```
1 Console.WriteLine("Dag " + "result" + " hoe gaat het met je?");
```

We krijgen dan altijd dezelfde output, namelijk:

```
1 Dag result hoe gaat het met je?
```

We tonen dus niet de inhoud van `result`, maar gewoon de tekst “result”.

1.3.10 Meer input vragen

Als je meerdere inputs van de gebruiker wenst te bewaren zal je meerdere geheugenplekken (variabelen) nodig hebben. Bijvoorbeeld:

```
1 Console.WriteLine("Geef leeftijd");
2 string leeftijd; //eerste variabele aanmaken
3 leeftijd = Console.ReadLine();
4 Console.WriteLine("Geef adres");
5 string adres; //tweede variabele aanmaken
6 adres = Console.ReadLine();
```

Je mag echter ook de variabelen al vroeger aanmaken. In C# zet men de geheugenplek creatie zo dicht mogelijk bij de code waar je die variabele gebruikt. Maar dat is geen verplichting. Dit mag dus ook:

```
1 string leeftijd; //eerste variabele aanmaken
2 string adres; //tweede variabele aanmaken
3 Console.WriteLine("Geef leeftijd");
4 leeftijd = Console.ReadLine();
5 Console.WriteLine("Geef adres");
6 adres = Console.ReadLine();
```



Je zal vaak `Console.WriteLine` moeten schrijven als je dit boek volgt. Ik heb echter goed nieuws voor je: er zit een ingebouwde *snippet* in VS om sneller `Console.WriteLine` op het scherm te toveren. Ik ga je niet langer in spanning houden... of toch... nog even. Ben je benieuwd? Spannend he!

Hier gaan we: **cw [tab] [tab]**

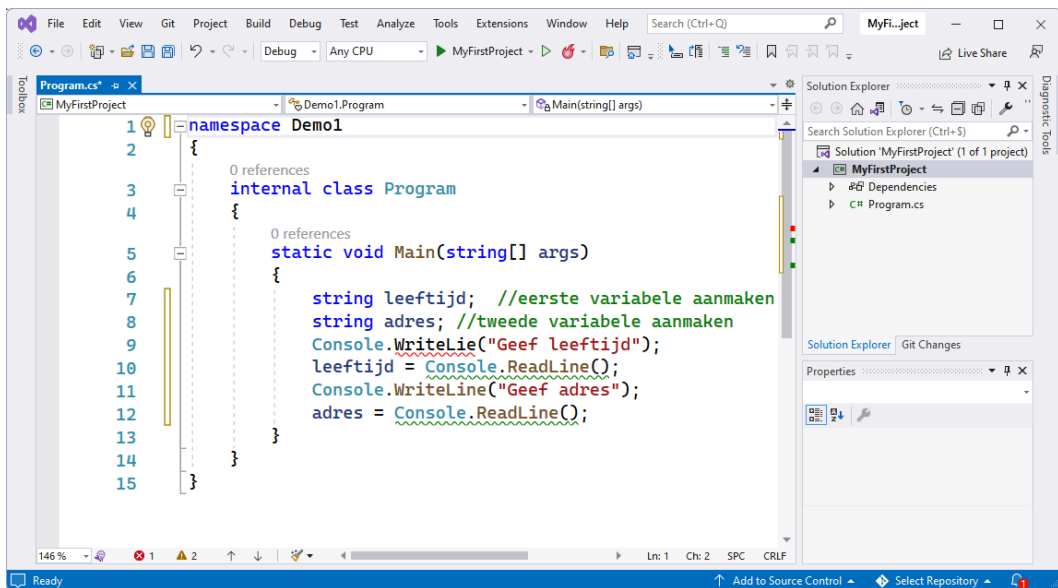
Als je dus `cw` schrijft en dan twee maal op de tab-toets van je toetsenbord duwt verschijnt daar *automagisch* een verse lijn met `Console.WriteLine();`.

1.4 Fouten oplossen

Je code zal pas compileren indien deze foutloos is geschreven. Herinner je dat computers uiterst dom zijn en dus vereisen dat je code 100% foutloos is qua woordenschat en grammatica.

Zolang er dus fouten in je code staan moet je deze eerst oplossen voor je verder kan. Gelukkig helpt VS je daarmee op 2 manieren:

- Fouten in code worden met een rode squiggly onderlijnd.
- Onderaan zie je in de statusbalk of je fouten hebt.



Figuur 1.16: Zie je de fout?

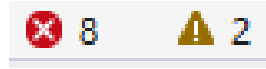
Laat je trouwens niet afschrikken door de gigantische reeks fouten die soms plots op je scherm verschijnen. VS begint al enthousiast fouten te zoeken terwijl je mogelijk nog volop aan het typen bent.



Als je plots veel fouten krijgt, kijk dan altijd vlak boven de plek waar de fouten verschijnen. Heel vaak zit daar de echte fout: meestal is dat gewoon het ontbreken van een komma-punt aan het einde van een statement.

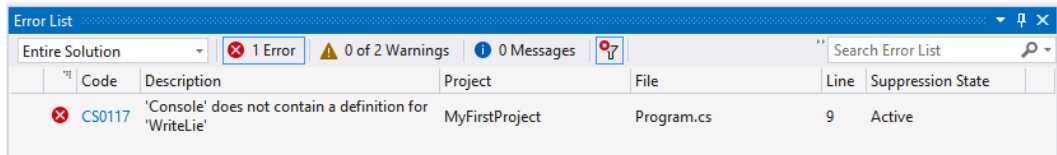
1.4.1 Fouten sneller vinden

Uiteraard ga je vaak code hebben die meerdere schermen omvat. Je kan via de error-list snel naar al je fouten gaan. Open deze door op het error-icoontje onderaan te klikken:



Figuur 1.17: So many errors?!

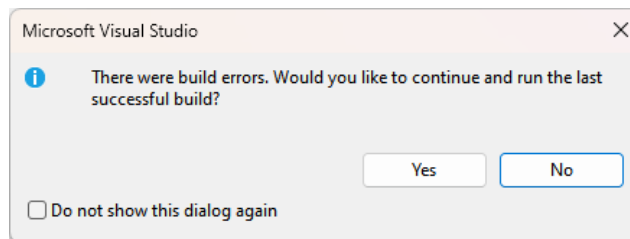
Dit zal de “error list” openen (**een schermdeel van VS dat ik aanraad om altijd open te laten én dus niet weg te klikken**). Warnings kunnen we - voorlopig - meestal negeren en deze ‘filter’ hoeft je dus niet aan te zetten.



Figuur 1.18: De error list.

In de error list kan je nu op iedere foutboodschap klikken om ogenblikkelijk naar de correcte lijn te gaan.

Zou je toch willen compileren en je hebt nog fouten dan zal VS je proberen tegen te houden. **Lees nu onmiddellijk wat de voorman hierover te vertellen heeft.**



Figuur 1.19: OPLETTEN!



Opletten aub : Indien je op de groene start knop duwt en bovenstaande waarschuwing krijgt **KLIK DAN NOOIT OP YES EN DUID NOOIT DE CHECKBOX AAN!**

Lees de boodschap eens goed na: wat denk je dat er gebeurt als je op ‘yes’ duwt? Inderdaad, VS zal de laatste werkende versie uitvoeren en dus niet de code die je nu hebt staan waarin nog fouten staan.

1.4.2 Fouten oplossen met lampje

Wanneer je je cursor op een lijn met een fout zet dan zal je soms vooraan een geel error-lampje zien verschijnen (dit duurt soms even):



Figuur 1.20: Lampje: de brenger der oplossingen... In tegenstelling tot Clippy de Office assistent uit de jaren '90....

Je kan hier op klikken en heel vaak krijg je dan ineens een mogelijke oplossing. **Wees steeds kritisch** hierover want VS is niet alwetend en kan niet altijd raden wat je bedoelt. Neem dus het voorstel niet zomaar over zonder goed na te denken of het dat was wat je bedoelde.



Warnings kan je voorlopig over het algemeen negeren . Bekijk ze gewoon af en toe. Wie weet bevatten ze nuttige informatie om je code te verbeteren.

1.4.3 Meest voorkomende fouten

De meest voorkomende fouten die je als beginnende C# programmeur maakt zijn:

- **Puntkomma** vergeten.
- **Schrijffouten** in je code, bijvoorbeeld `RaedLine` i.p.v. `ReadLine`.
- Geen rekening gehouden met **hoofdletter gevoeligheid van C#**, bijvoorbeeld `ReadLine` i.p.v. `ReadLine` (zie verder).
- Per ongeluk **accolades verwijderd**.
- Code geschreven op plekken waar dat niet mag (je mag momenteel enkel binnen de accolades van `Main` schrijven).

1.5 Kleuren in console

Je kan in console-applicaties zelf bepalen in welke kleur nieuwe tekst op het scherm verschijnt. Je kan zowel de **kleur van het lettertype** instellen (via `ForegroundColor`) als de **achtergrond-kleur** (`BackgroundColor`).

Je kan met de volgende expressies de console-kleur veranderen, bijvoorbeeld de achtergrond in blauw en de letters in groen:

```
1 Console.BackgroundColor = ConsoleColor.Blue;  
2 Console.ForegroundColor = ConsoleColor.Green;
```

Vanaf dan zal alle tekst die je hierna met `WriteLine` en `Write` naar het scherm stuurt met deze kleuren werken. Merk op dat we **bestaande tekst op het scherm níét van kleur kunnen veranderen zonder deze eerst te verwijderen en dan opnieuw, met andere kleurinstellingen, naar het scherm te sturen**.



Alle kleuren die beschikbaar zijn staan beschreven in `ConsoleColor` deze zijn: Black, DarkBlue, DarkGreen, DarkCyan, DarkRed, DarkMagenta, DarkYellow, Gray, DarkGray, Blue, Green, Cyan, Red, Magenta, Yellow.

Wens je dus de kleur Red dan zal je deze moeten aanroepen door er `ConsoleColor` . voor te zetten: `ConsoleColor.Red`.

Waarom is dit? `ConsoleColor` is een zogenaamd **enum**-type. Enums leggen we verderop in hoofdstuk 5 uit.

Een voorbeeld:

```
1 Console.WriteLine("Tekst in de standaard kleur");  
2 Console.BackgroundColor = ConsoleColor.Yellow;  
3 Console.ForegroundColor = ConsoleColor.Black;  
4 Console.WriteLine("Zwart met gele achtergrond");  
5 Console.ForegroundColor = ConsoleColor.Red;  
6 Console.WriteLine("Rood met gele achtergrond");
```

Als je deze code uitvoert krijg je als resultaat:

```
Tekst in de standaard kleur  
Zwart met gele achtergrond  
Rood met gele achtergrond
```

Figuur 1.21: Resultaat voorgaande code.



Kleur in console gebruiken is nuttig om je gebruikers een minder eentonig en meer informatieve applicatie aan te bieden. Je zou bijvoorbeeld alle foutmeldingen in het rood kunnen laten verschijnen. Let er wel op dat je applicatie geen aartslelijk programma wordt.

Hou er ook rekening mee dat niet iedereen (alle) kleuren kan zien. In de vorige editie van dit boek gebruikte ik rode letters op een groene achtergrond. Dat resulteerde in onleesbare tekst voor mensen met *Daltonisme*.

1.5.1 Kleur resetten

Soms wil je terug de originele applicatie-kleuren hebben. Je zou manueel dit kunnen instellen, maar wat als de gebruiker slechtiend is en in z'n besturingssysteem andere kleuren als standaard heeft ingesteld?!

De veiligste manier is daarom de kleuren te resetten door de `Console.ResetColor()` methode aan te roepen zoals volgend voorbeeld toont:

```
1 Console.ForegroundColor = ConsoleColor.Red;
2 Console.WriteLine("Error!!!! Contacteer de helpdesk");
3 Console.ResetColor();
4 Console.WriteLine("Het programma sluit nu af");
```

1.6 Waar zijn de oefeningen?!

Huh?! Waar zijn de oefeningen naartoe die de vorige edities van dit handboek nog wel hadden? Om bomen te besparen heb ik besloten om alle oefeningen via **ziescherp.be** beschikbaar te stellen. Je zal langs die webpagina een grote verzamelingen oefeningen vinden, die op de koop toe geregeld vernieuwd en verbeterd worden.

Je kan trouwens gratis op Quizlet deze cursus dagelijks instuderen⁵, de ideale manier om snel essentiële C# begrippen voor altijd te onthouden.



Sinds 2023 is er een gigantische opkomst van nog straffere A.I. tools, met ChatGPT voorop. Alhoewel deze tools vaak heel goede C# code kunnen genereren, raden we af deze te gebruiken, om dezelfde redenen dat je best IntelliCode niet gebruikt (zie hoofdstuk 7). Vraag daarom nooit aan ChatGPT om “oefening x” voor je op te lossen. Moet je dan ChatGPT volledig links laten liggen? Uiteraard niet. Gebruik hem als extra leermiddel om bijvoorbeeld stukken code toe te lichten, bepaalde concepten op een andere manier uit te leggen enz.

⁵Via <https://quizlet.com/join/mqzQCGJCF>.

2 De basisconcepten van C#

Om een werkend C#-programma te maken moeten we de C#-taal beheersen. Net zoals iedere taal bestaat ook C# uit:

- grammatica: in de vorm van de **C# syntax**
- woordenschat: in de vorm van gereserveerde **keywords**.

Een C#-programma bestaat uit een opeenvolging van instructies, **statements** genoemd. **Statements eindigen steeds met een puntkomma**. Net zoals ook in het Nederlands een zin meestal eindigt met een punt. Ieder statement kan je vergelijken als één lijn in ons recept, het algoritme.

De volgorde van de woorden in C# zijn niet vrijblijvend en moeten aan grammaticale regels voldoen (de syntax). Enkel indien alle statements correct zijn zal het programma gecompileerd worden naar een werkend programma.

Enkele belangrijke regels van C#:

- **Hoofdlettergevoelig:** C# is hoofdlettergevoelig. Dat wil zeggen dat hoofdletter R en kleine letter r totaal verschillende zaken zijn voor C#. Reinhardt en reinhardt zijn dus ook niet hetzelfde.
- **Statements afsluiten met puntkomma (; **):** Doe je dat niet dan zal C# denken dat de regel gewoon op de volgende lijn doorloopt en zal deze dan als één (fout) geheel proberen te compileren.
- **Witruimtes:** Spaties, tabs en enters worden door de C# compiler genegeerd. Je kan ze dus gebruiken om de layout van je code (*bladspiegel*) te verbeteren. De enige plek waar witruimtes wél een verschil geven is tussen aanhalingstekens " " die we later zullen leren gebruiken.
- **Commentaar toevoegen kan:** door // voor een enkele lijn te zetten zal deze lijn genegeerd worden door de compiler. Je kan ook meerdere lijnen code in commentaar zetten door er /* voor en */ achter te zetten.
- **Je code begint altijd in de Main-methode!!!**
- **Van boven naar onder:** je code wordt van boven naar onder uitgevoerd en zal enkel naar andere plaatsen springen als je daar expliciet in je code om vraagt.

2.1 Keywords: de woordenschat

C# bestaat zoals gezegd niet enkel uit grammaticale regels. Grammatica zonder woordenschat is nutteloos. Er zijn binnen C# dan ook momenteel 80 woorden, zogenaamde **reserved keywords** die de woordenschat voorstellen. Het spreekt voor zich dat deze keywords een eenduidige, specifieke betekenis hebben en dan ook enkel voor dat doel gebruikt kunnen worden.

In dit boek zullen we stelselmatig deze keywords leren kennen en gebruiken op een correcte manier om zo werkende code te maken.

Deze keywords zijn:

| | | | |
|------------------|---------------------|------------------|------------------|
| <i>abstract</i> | <i>as</i> | <i>base</i> | bool |
| break | byte | case | <i>catch</i> |
| char | <i>checked</i> | <i>class</i> | const |
| <i>continue</i> | decimal | <i>default</i> | <i>delegate</i> |
| do | double | else | enum |
| <i>event</i> | <i>explicit</i> | <i>extern</i> | false |
| <i>finally</i> | <i>fixed</i> | float | for |
| <i>foreach</i> | <i>goto</i> | if | <i>implicit</i> |
| <i>in</i> | int | <i>interface</i> | <i>internal</i> |
| <i>is</i> | <i>lock</i> | long | namespace |
| <i>new</i> | <i>null</i> | <i>object</i> | <i>operator</i> |
| out | <i>override</i> | <i>params</i> | <i>private</i> |
| <i>protected</i> | <i>public</i> | <i>readonly</i> | ref |
| return | sbyte | <i>sealed</i> | short |
| <i>sizeof</i> | <i>stackalloc</i> | <i>static</i> | string |
| <i>struct</i> | switch | <i>this</i> | <i>throw</i> |
| true | <i>try</i> | <i>typeof</i> | uint |
| ulong | <i>unchecked</i> | <i>unsafe</i> | ushort |
| <i>using</i> | <i>using static</i> | <i>virtual</i> | void |
| <i>volatile</i> | while | | |

De keywords in **vet** zijn keywords die we in het eerste deel van dit boek zullen bekijken (hoofdstukken 1 tot en met 8). Die in **cursief** in het tweede deel (9 en verder). De overige zal je zelf moeten ontdekken ... of mogelijk zelfs nooit in je carrière gebruiken vanwege hun soms obscure nut.



C# is een levende taal. Soms verschijnen er dan ook nieuwe keywords. De afspraak is echter dat de lijst hierboven niet verandert. Nieuwe keywords maken deel uit van de *contextual keywords* en zullen nooit gereserveerde keywords worden. We zullen enkele van deze “nieuwere” keywords tegenkomen waaronder: **get**, **set**, **value** en **var**.



*Aandacht, aandacht! Step away from the keyboard! I repeat. Step away from the keyboard. Hierbij wil ik u attent maken op een belangrijke, onbeschreven, wet voor C# programmeurs: “**NEVER EVER USE GOTO**”*

*Het moet hier alvast even uit m’n systeem. **goto** is weliswaar een officieel C# keyword, toch zal je het in dit boek **nooit** zien terugkomen in code. Je kan alle problemen in je algoritmes oplossen zonder ooit **goto** nodig te hebben.*

*Voel je toch de drang: **don’t!** Simpelweg, don’t. Het is het niet waard. Geloof me.*

NEVER USE GOTO.

Enneuh, ik hou je in’t oog hoor!

2.2 Variabelen, identifiers en naamgeving

Variabelen zijn nodig om tijdelijke data in op te slaan, zoals gebruikersinput, zodat we deze later in het programma kunnen gebruiken.

We doen hetzelfde in ons hoofd wanneer we bijvoorbeeld zeggen “tel 3 en 4 op en vermenigvuldig dat resultaat met 5”. Eerst zullen we het resultaat van “3+4” in een variabele moeten bewaren. Vervolgens zullen we de inhoud van die variabele vermenigvuldigen met 5 en dat nieuwe resultaat ook in een nieuwe variabele opslaan.

Wanneer we een variabele aanmaken, zal deze moeten voldoen aan enkele afspraken. Zo moeten we minstens 2 zaken meegeven:

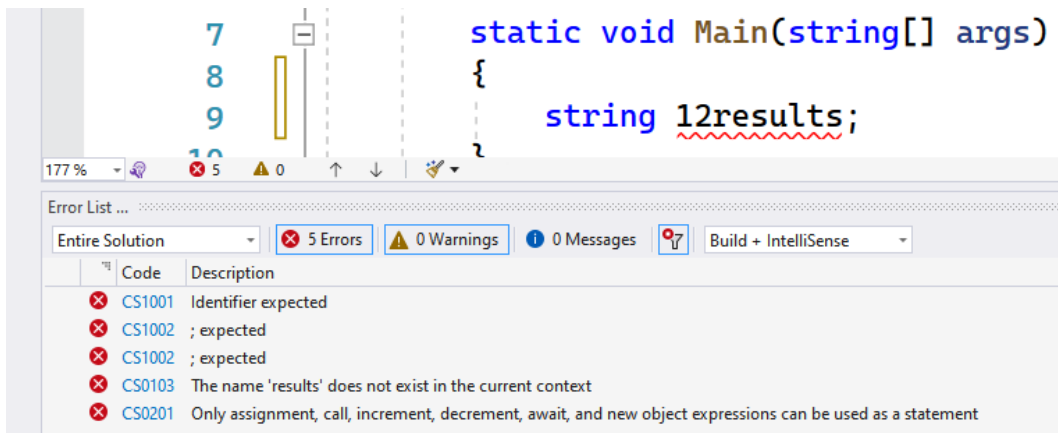
- De **identifier** waarmee we snel aan de variabele-waarde kunnen. Dit is de gebruiksvriendelijke naam die we geven aan een geheugenplek.
- Het **datatype** dat aangeeft wat voor soort data we wensen op te slaan. Enkel en alleen dat soort type data zal in deze variabele kunnen bewaard worden.

2.2.1 Regels voor identifiers

De code die we gaan schrijven moet voldoen aan een hoop regels. Wanneer we in onze code zelf namen (**identifiers**) geven aan variabelen (en later ook methoden, objecten, enz.) dan moeten we een aantal regels volgen:

- **Hoofdlettergevoelig:** de identifiers `tim` en `Tim` zijn verschillend zoals reeds vermeld.
- **Geen keywords:** identifiers mogen geen gereserveerde C# keywords zijn. De keywords van 2 pagina's terug mogen dus niet. Varianten waarbij de hoofdletters anders zijn mogen wel. `gOTO` en `sTRING` mogen dus wel, maar niet `goto` of `string` want dat zijn gereserveerde keywords. Een ander voorbeeld `INT` mag bijvoorbeeld wel, maar `int` niet.
- **Eerste karakter-regel:** het eerste karakter van de identifier mag een **kleine of grote letter**, of een **liggend streepje** (`_`) zijn.
- **Alle andere karakters-regels:** de overige karakters volgende de eerste karakter-regel, maar mogen ook cijfers zijn.
- **Lengte:** Een legale identifier mag zo lang zijn als je wenst, maar je houdt het best leesbaar.

Volg je voorgaande regels niet dan zal je code niet gecompileerd worden en zal VS de identifiers in kwestie als een fout aanduiden. Of beter, als een hele hoop foutboodschappen. Schrik dus niet als je bijvoorbeeld het volgende ziet:



Figuur 2.1: Zoals je ziet raakt VS volledig de kluts kwijt als je je niet houdt aan de identifier regels.

2.2.1.1 Enkele voorbeelden

Enkele voorbeelden van toegelaten en niet toegelaten identifiers:

| Identifier | Toegelaten? | Uitleg indien niet toegelaten |
|----------------|-------------|---|
| werknemer | ja | |
| kerst2018 | ja | |
| pippo de clown | neen | geen spaties toegestaan |
| 4dPlaats | neen | mag niet starten met een cijfer |
| _ILOVE2022 | ja | |
| Tor+Bjorn | neen | enkel cijfers, letters en liggende streepjes toegestaan |
| ALLCAPSMAN | ja | |
| B_A_L | ja | |
| class | neen | gereserveerd keyword |
| WriteLine | ja | |
| _____ | ja | |

2.2.2 Naamgeving afspraken

Er zijn geen vaste afspraken over hoe je je variabelen moet noemen toch hanteren we enkele **coding richtlijnen**:

- **Duidelijke naam:** de identifier moet duidelijk maken waarvoor de identifier dient. Schrijf dus liever gewicht of leeftijd in plaats van a of meuh.
- **Camel casing:** gebruik camel casing indien je meerdere woorden in je identifier wenst te gebruiken. Camel casing wil zeggen dat ieder nieuw woord terug met een hoofdletter begint. Een goed voorbeeld kan dus zijn leeftijdTimDams of aantalLeerlingenKlas1EA. Merk op dat we liefst het eerste woord met kleine letter starten. Uiteraard zijn er geen spaties toegelaten.

2.3 Commentaar

Soms wil je misschien extra commentaar bij je code zetten. Op die manier kan je extra informatie aan jezelf of andere *lezers* van je code geven.

2.3.1 Enkele lijn commentaar

Eén lijn commentaar geef je aan door de lijn te starten met twee voorwaartse slashes `//`. Uiteraard mag je ook meerdere lijnen op deze manier in commentaar zetten. Zo wordt dit ook vaak gebruikt om tijdelijk een stuk code “uit te schakelen”. Ook mogen we commentaar *achter* een stuk C# code plaatsen zoals we hieronder tonen.

`//` zal alle tekens die volgen tot aan de volgende witregel in commentaar zetten:

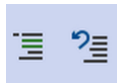
```
1 //De start van het programma
2 int getal = 3;
3 //Nu gaan we rekenen
4 int result = getal * 5;
5 // result = 3*5;
6 Console.WriteLine(result); //We tonen resultaat op scherm: 15
```

2.3.2 Blok commentaar

We kunnen een stuk tekst als commentaar aangeven door voor de tekst `/*` te plaatsen en `*/` achteraan. Een voorbeeld:

```
1 /*
2     Een blok commentaar
3     Een heel verhaal, dit wordt mooi
4     Is dit een haiku?
5 */
6 int leeftijd = 0;
```

Je kan ook code in VS selecteren en dan met de comment/uncomment-knoppen in de menubalk heel snel lijnen of hele blokken code van commentaar voorzien, of deze net weghalen:



Figuur 2.2: De linkse knop voegt comment tags toe, de andere verwijdert ze.

2.4 Datatypes

Een essentieel onderdeel van C# is kennis van datatypes. Binnen C# zijn een aantal types gedefinieerd die je kan gebruiken om data in op te slaan. Wanneer je data wenst te bewaren in je applicatie dan zal je je moeten afvragen wat voor soort data het is. Gaat het om een geheel getal, een kommagetal, een stuk tekst of misschien een binaire reeks? Ieder datatype in C# kan één welbepaald soort data bewaren en dit zal telkens een bepaalde hoeveelheid computergeheugen vereisen.



Datatypes zijn een belangrijk concept in C# omdat deze taal een zogenaamde “**strongly typed language**” is (in tegenstelling tot bijvoorbeeld JavaScript). Wanneer je in C# data wenst te bewaren (in een variabele) zal je van bij de start moeten aangeven wat voor data dit zal zijn. Vanaf dan zal de data op die geheugenplek op dezelfde manier verwerkt worden en niet zo maar van ‘vorm’ kunnen veranderen zonder extra input van de programmeur.

Bij JavaScript kan dit bijvoorbeeld wel, wat soms een fijn werken is, maar ook vaak vloeken: je bent namelijk niet gegarandeerd dat je variabele wel het juiste type zal bevatten wanneer je het gaat gebruiken.

Er zijn verscheidene basistypes in C# gedeclareerd, zogenaamde **primitieve datatypes**:

In dit boek leren we werken met datatypes voor:

- Gehele getallen: **sbyte**, **byte**, **short**, **ushort**, **int**, **uint**, **long**, **ulong**
- Kommagetallen: **double**, **float**, **decimal**
- Tekst: **char**, **string**
- Booleans: **bool**
- Enums (een speciaal soort datatype dat een beetje een combinatie van meerdere datatypes is én dat je zelf deels kan definiëren.)

Ieder datatype wordt gedefinieerd door minstens volgende eigenschappen:

- **Soort data** dat in de variabele van dit type kan bewaard worden (tekst, geheel getal, enz.)
- **Geheugengrootte**: de hoeveelheid bits dat 1 element van dit datatype inneemt in het geheugen. Dit kan belangrijk zijn wanneer je met véél data gaat werken en je niet wilt dat de gebruiker drie miljoen gigabyte RAM nodig heeft.
- **Schrijfwijze van de literals**: hoe weet C# of 2 een komma getal (2.0) of een geheel getal (2) is? Hiervoor gebruiken we specifieke schrijfwijzen van deze waarden (**literals**) wat we verderop uiteraard uitgebreid zullen bespreken.



Het datatype **string** heb je al gezien in het vorig hoofdstuk. Je hebt toen een variabele aangemaakt van het type string door de zin **string** result;. Verderop plaatsen we dan iets waar de gebruiker iets kan intypen in die variabele:

```
1 result = Console.ReadLine();
```

2.4.1 Basistypen voor getallen

Alhoewel een computer digitaal werkt en enkel 0'n en 1'n bewaart zou dat voor ons niet erg handig werken. C# heeft daarom een hoop datatypes gedefinieerd om te werken met getallen zoals wij ze kennen, gehele en kommagetallen. Intern zullen deze getallen nog steeds binair bewaard worden, maar dat is tijdens het programmeren zelden een probleem.

De basistypen van C# om getallen in op te slaan zijn:

- Voor gehele getallen: **sbyte**, **byte**, **short**, **int**, **long** en **char**.
- Voor natuurlijke getallen (enkel positief): **ushort**, **uint** en **ulong**.
- Voor kommagetallen: **double**, **float** en **decimal**.

Deze datatypes hebben allemaal een verschillend bereik, wat een rechtstreekse invloed heeft op de hoeveelheid geheugen die ze innemen.



Ieder type hierboven heeft een bepaald bereik en hoeveelheid geheugen nodig. Je zal dus steeds moeten afwegen wat je wenst. Op een high-end pc met vele gigabytes aan werkgeheugen (RAM) is geheugen zelden een probleem waar je rekening mee moet houden.

Of toch: wat met real-time first person shooters die miljoenen berekeningen per seconde moeten uitvoeren? Daar zal iedere bit en byte tellen. Op andere apparaten (smartphone, arduino, smart fridges, enz.) is iedere byte geheugen nog kostbaarder.

Kortom: kies steeds bewust het datatype dat het beste 'past' voor je probleem qua bereik, precisie en geheugengebruik.

2.4.1.1 Gehele getallen

Voor de gehele getallen zijn er volgende datatypes:

| Type | Geheugen | Bereik (waardenverzameling) |
|---------------|----------|--|
| sbyte | 8 bits | -128 tot 127 |
| byte | 8 bits | 0 tot 255 |
| short | 16 bits | -32 768 tot 32 767 |
| ushort | 16 bits | 0 tot 65535 |
| int | 32 bits | -2 147 483 648 tot 2 147 483 647 |
| uint | 32 bits | 0 tot 4 294 967 295 |
| long | 64 bits | -9 223 372 036 854 775 808 tot 9 223 372 036 854 775 807 |
| ulong | 64 bits | 0 tot 18 446 744 073 709 551 615 |
| char | 16 bits | 0 tot 65 535 |

Het bereik van ieder datatype is een rechtstreeks gevolg van het aantal bits waarmee het getal in dit type wordt voorgesteld. De **short** bijvoorbeeld wordt voorgesteld door 16 bits. Eén bit daarvan wordt gebruikt voor het teken (0 of 1, + of -). De overige 15 bits worden gebruikt voor de waarde: van 0 tot $2^{15}-1$ (= 32767) en van -1 tot -2^{15} (= -32768)

Enkele opmerkingen bij voorgaande tabel:

- De s vooraan **sbyte** staat voor *s*igned: m.a.w. 1 bit wordt gebruikt om het + of - teken te bewaren.
- De u vooraan **ushort**, **uint** en **ulong** staat voor *u*nsigned. Het omgekeerde van signed dus. Kwestie van het ingewikkeld te maken. Deze twee datatypes hebben dus geen teken en zijn **altijd positief**.
- **char** bewaart karakters. We zullen verderop dit datatype uitspitten en ontdekken dat karakters (alle tekens op het toetsenbord, inclusief getallen, leesteken, enz.) als gehele, binaire getallen worden bewaard. Daarom staat **char** in deze lijst.
- Het grootste getal bij **long** is $2^{63}-1$ (*negen triljoen tweehonderddrieëntwintig biljard driehonderd tweeënzeventig biljoen zesendertig miljard achthonderdvierenvijftig miljoen zeventighonderdvijfenzeventigduizend achthonderd en zeven*). Dit zijn maar 63 bits?! Inderaad, de laatste bit wordt wederom gebruikt om het teken te bewaren.



“Wow. Moet je al die datatypes uit het hoofd kennen? Ik was al blij dat ik tekst op het scherm kon tonen.”

Uiteraard kan het geen kwaad dat je de belangrijkste datatypes onthoudt, anderzijds zul je zelf merken dat door gewoon veel te programmeren je vanzelf wel zult ontdekken welke datatypes je waar kunt gebruiken. Laat je dus niet afschrikken door de ellenlange tabellen met datatypes in dit hoofdstuk, we gaan er maar een handvol effectief van gebruiken.

2.4.1.2 Kommagetallen

Voor de kommagetallen zijn er maar 3 mogelijkheden. Ieder datatype heeft een ‘voordeel’ tegenover de 2 andere, dit voordeel staat vet in de tabel:

| Type | Geheugen | Bereik | Precisie |
|----------------|----------------|---------------|---------------------|
| float | 32 bits | gemiddeld | ~6-9 digits |
| double | 64 bits | meeste | ~15-17 digits |
| decimal | 128 bits | minste | 28-29 digits |

Zoals je ziet moet je bij kommagetallen een afweging maken tussen 3 even belangrijke criteria. Heb je ongelooflijk grote precisie nodig dan ga je voor een **decimal**. Wil je vooral erg grote of erg kleine getallen kies je voor **double**. Zoals je merkt zal je dus zelden **decimal** nodig hebben, deze zal vooral nuttig zijn in financiële en wetenschappelijke programma's waar met erg exacte cijfers moet gewerkt worden.



Bij twijfel opteren we meestal voor kommagetallen om het **double** datatype te gebruiken. Bij gehele getallen kiezen we meestal voor **int**.



De precisie van een getal is het aantal beduidende cijfers. Enkele voorbeelden:

- 2.2345 heeft precisie 5.
- 2.23 heeft precisie 3.
- 0.0032 heeft precisie 2.

2.4.2 Boolean datatype

bool (**boolean**) is het eenvoudigste datatype van C#. Het kan maar 2 mogelijke waarden bevatten: **true** of **false**. 0 of 1 met andere woorden.

We zullen het **bool** datatype erg veel nodig hebben wanneer we met beslissingen zullen werken in een later hoofdstuk, specifiek de **if** statements die afhankelijk van de waarde van een **bool** bepaalde code wel of niet zullen doen uitvoeren.



Het gebeurt vaak dat beginnende programmeurs een **int** variabele gebruiken terwijl ze toch weten dat de variabele maar 2 mogelijke waarden zal hebben. Om dus geen onnodig geheugen te verbruiken is het aan te raden om in die gevallen steeds met een **bool** variabele te werken.



Het **bool** datatype is uiteraard het kleinst mogelijke datatype. Hoeveel geheugen zal een variabele van dit type innemen denk je? Inderdaad **1 bit**.

2.4.3 Tekst/String datatype

Ik besteed verderop een heel apart hoofdstuk om te tonen hoe je één enkel karakter of volledige flarden tekst kan bewaren in variabelen.

Hier alvast een voorsmaakje:

- Tekst kan bewaard worden in het **string** datatype.
- Een enkel karakter wordt bewaard in het **char** datatype dat we ook hierboven al even hebben zien passeren.



*Wat een gortdroge tekst was me dat nu net? Waarom moeten we al deze datatypes kennen? Wel, we hebben deze nodig om **variabelen** aan te maken. En variabelen zijn het hart van ieder programma. Zonder variabelen ben je aan het programmeren aan een programma dat een soort vergevorderde vorm van dementie heeft en hoegenaamd niets kan onthouden.*

2.5 Variabelen

De data die we in een programma gebruiken bewaren we in een **variabele van een bepaald datatype**. Een variabele is een plekje in het geheugen dat in je programma zal gereserveerd worden om daarin data te bewaren van het type dat je aan de variabele hebt toegekend.

Een variabele heeft een geheugenadres, namelijk de plek waar de data in het geheugen staat. Maar het zou lastig programmeren zijn indien je steeds dit adres moest gebruiken. Daarom moeten we ook steeds een naam oftewel **identifier** aan de variabele geven. Op die manier kunnen we eenvoudig de geheugenplek aanduiden en hoeven we niet te werken met een lang hexadecimaal geheugen adres (bv. 0x4234FE13EF1).



De identifier van de variabele moet uiteraard voldoen aan de *identifier regels* zoals eerder besproken.

2.5.1 Variabelen aanmaken en gebruiken

Om een variabele te maken moeten we deze **declareren**, door een type en naam te geven. Vanaf dan zal de computer een hoeveelheid geheugen voor je reserveren waar de inhoud van deze variabele in kan bewaard worden. Hiervoor dien je minstens op te geven:

1. Het **datatype** (bv. **int**, **double**).
2. Een **identifier** zodat de variabele uniek kan geïdentificeerd worden volgens de naamgevingsregel van C#.
3. (optioneel) Een **beginwaarde** die de variabele krijgt bij het aanmaken ervan.

Een variabele declaratie heeft als syntax:

```
1 datatype identifier;
```

Enkele voorbeelden:

```
1 int leeftijd;  
2 string leverAdres;  
3 bool isGehuwd;
```

Indien je weet wat de beginwaarde moet zijn van de variabele dan mag je de variabele ook reeds deze waarde toekennen bij het aanmaken:

```
1 int mijnLeeftijd = 37;
```




Je mag ook meerdere variabelen van het zelfde datatype in 1 enkele declaratie aanmaken door deze met komma's te scheiden:

```
1 datatype identifier1, identifier2, identifier3;
```

Bijvoorbeeld **string** voornaam, achternaam, adres;

2.5.2 Waarden toekennen aan variabelen

Van zodra je een variabele hebt gedeclareerd kunnen we dus ten allen tijde deze variabele gebruiken om een waarde aan toe te kennen, de bestaande waarde te overschrijven, of de waarde te gebruiken, zoals:

- **Waarde toekennen:** Herinner dat de toekenning steeds gebeurt van rechts naar links: het deel rechts van het gelijkheidsteken wordt toegewezen aan het deel links er van, bijvoorbeeld: `mijnGetal = 15;`
- **Waarde gebruiken:** Bijvoorbeeld `anderGetal = mijnGetal + 15;`
- **Waarde tonen op scherm:** Bijvoorbeeld `Console.WriteLine(mijnGetal);`

Met de **toekenning-operator (=)** kan je een waarde toekennen aan een variabele. Hierbij kan je zowel een literal toekennen oftewel het resultaat van een expressie .

Je kan natuurlijk ook een waarde uit een variabele uitlezen en toewijzen (kopiëren) aan een andere variabele:

```
1 int eenAndereLeeftijd = mijnLeeftijd;
```

2.5.3 Literals

Literals zijn expliciet neergeschreven waarden in je code. De manier waarop je een literal schrijft in je code zal bepalen wat het datatype van die literal is:

- **Gehele getallen** worden standaard als **int** beschouwd, vb: 125.
- **Kommagetallen** (met punt .) worden standaard als **double** beschouwd, vb: 12.5.

Wil je echter andere getaltypes dan **int** of **double** een waarde geven dan moet je dat dus expliciet in de literal aanduiden. Hiervoor plaats je een *suffix* achter de literalwaarde. Afhankelijk van deze suffix duidt je dan aan om welke datatype het gaat:

- U of u voor **uint**, vb: 125U (dus bijvoorbeeld `aantalSchapen = 27u;`)
- L of l voor **long**, vb: 125L.
- UL of ul voor **ulong**, vb: 125ul.
- F of f voor **float**, vb: 12.5f.
- M of m voor **decimal**, vb: 12.5M.

Naast getallen zijn er uiteraard ook nog andere datatypes waar we de literals van moeten kunnen schrijven:

Voor **bool** zijn dit enkel **true** en **false**.

Voor **char** wordt dit aangeduid met een enkele apostrof voor en na de literal. Denk maar aan **char** laatsteLetter = 'z';.

Voor **string** wordt dit aangeduid met aanhalingsteken voor en na de literal. Bijvoorbeeld **string** myPoke = "pikachu".



Om samen te vatten, even de belangrijkste literal schrijfwijzen op een rijtje:

```
1 int getal = 5;
2 double anderGetal = 5.5;
3 uint nogAnderGetal = 15u;
4 float kleinKommaGetal = 158.9f;
5 char letter = 'k';
6 bool isDitCool = true;
7 string zin = "Ja hoor";
```

De overige types **sbyte**, **short** en **ushort** hebben geen literal aanduiding. Er wordt vanuit gegaan wanneer je een literal probeert toe te wijzen aan één van deze datatypes dat dit zonder problemen zal gaan (ze worden impliciet geconverteerd).

Volgende code mag dus:

```
1 sbyte start = 127;
```

Dit wordt toegestaan, de **int** literal 127 zal geconverteerd worden achter de schermen naar een **sbyte** en dan toegewezen worden.

2.5.3.1 Literal toewijzen

Als je in je code expliciet de waarde 4 wilt toekennen aan een variabele dan is het getal 4 in je code een zogenaamde **literal**.

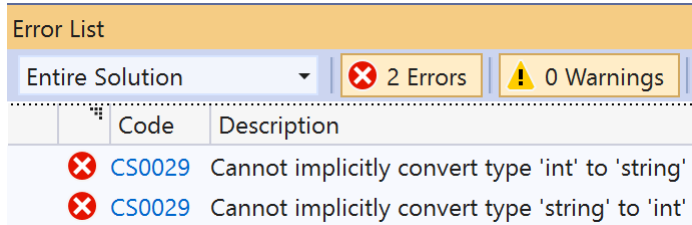
Voorbeelden van een literal toekennen:

```
1 int temperatuurGisteren = 20; //20 is de literal
2 int temperatuurVandaag = 25; //25 is de literal
```

Het is belangrijk dat het type van de literal overeenstemt met dat van de variabele waaraan je deze zal toewijzen. Volgende code zal dan ook een compiler-fout genereren. Je probeert een **string**-literal aan een **int**-variabele wil toewijzen, en omgekeerd:

```
1 string eenTekst;
2 int eenGetal;
3 eenTekst = 4;
4 eenGetal = "4";
```

Als je bovenstaande probeert te compileren dan krijg je volgende foutboodschappen:



Figuur 2.3: Foutboodschap wanneer je literals toekent van een verkeerd datatype.

2.5.3.1.1 Hexadecimale en binaire notatie Je kan ook hexadecimale notatie (starten met 0x of 0X) gebruiken wanneer je bijvoorbeeld met **int** of **byte** werkt:

```
1 int mijnLeeftijd = 0x0024; //36
2 byte mijnByteWaarde = 0x00C9; //201
```

Ook binaire notatie (starten met 0b of 0B) kan:

```
1 int mijnLeeftijd = 0b001001000; //72
2 int andereLeeftijd = 0b0001_0110_0011_0100_0010 //idem, maar met _
als separator
```

Deze schrijfwijzen kunnen handig zijn wanneer je met binaire of hexadecimale data wilt werken.

2.5.3.2 Beginwaarden van variabelen

Het is een goede gewoonte om variabelen steeds ogenblikkelijk een beginwaarde toe te wijzen. Alhoewel C# altijd vers gedeclareerde variabelen een standaard beginwaarde zal geven, is dit niet zo in oudere programmeertalen. In sommige talen zal een variabele een volledig willekeurige beginwaarde krijgen. Gelukkig in C# is dat niet, maar geef toch maar direct steeds een waarde, al was het maar om je literals te oefenen.

De standaard beginwaarde van een variabele hangt natuurlijk van het datatype af:

- Voor getallen is dat steeds de nulwaarde (dus 0 bij **int**, 0.0 bij **double**, enz.).
- Bij variabelen van het type **bool** is dat **false**.
- Bij **char** is dat de literal: `\0` (in het volgende hoofdstuk leggen we die vreemde backslash uit).
- En bij tekst is dat de lege **string**-literal: `""` (maar je mag ook `String.Empty` gebruiken).

2.5.4 Nieuwe waarden overschrijven oude waarden

Wanneer je een reeds gedeclareerde variabele een **nieuwe waarde toekent** dan zal de oude waarde in die variabele onherroepelijk verloren zijn. Probeer dus altijd goed op te letten of je de oude waarde nog nodig hebt of niet. Wil je de oude waarde ook nog bewaren dan zal je een nieuwe, extra variabele moeten aanmaken en daarin de nieuwe waarde moeten bewaren:

```
1 int temperatuurGisteren = 20;  
2 temperatuurGisteren = 25;
```

In dit voorbeeld zal er voor gezorgd worden dat de oude waarde van `temperatuurGisteren` (20) overschreven zal worden met 25.

Volgende code toont hoe je bijvoorbeeld eerst de vorige waarde kunt bewaren en dan overschrijven:

```
1 int temperatuurGisteren = 20;  
2 //Doe van alles  
3 //...  
4 //Vervolgens: vorige temperatuur in eergisteren bewaren  
5 int temperatuurEerGisteren = temperatuurGisteren;  
6 //temperatuur nu overschrijven  
7 temperatuurGisteren = 25;
```

We hebben aan het einde van het programma zowel de temperatuur van eergisteren (20), als die van gisteren (25).



Een veel gemaakte fout is variabelen meer dan één keer declareren. Dit mag niet! Van zodra je een variabele declareert is deze bruikbaar in de scope (zie hoofdstuk 5) tot het einde. Binnen de scope van die variabele kan je geen nieuwe variabele aanmaken met dezelfde naam (zelfs niet wanneer het type anders is).

Volgende code zal dus een fout geven:

```
1 double kdRating = 2.1;  
2 //even later...  
3 double kdRating = 3.4;
```

De foutboodschap vertelt duidelijk wat het probleem is: *A local variable or function named 'kdRating' is already defined in this scope.*

Lijn 3 moet dus worden:

```
1 kdRating = 3.4;
```

2.6 Expressies en operators

Zonder expressies is programmeren saai: je kan dan enkel variabelen aan elkaar toewijzen. Expressies zijn als het ware eenvoudige tot complexe sequenties van bewerkingen die op 1 resultaat uitkomen met een specifiek datatype. De volgende code is bijvoorbeeld een expressie: $3+2$.

Het resultaat van deze expressie is **5** (type **int**).

2.6.1 Expressie-resultaat toewijzen

Meestal zal je expressies schrijven waarin je bewerkingen op en met variabelen uitvoert. Vervolgens zal je het resultaat van die expressie willen bewaren voor verder gebruik in je code. In de volgende code kennen we het **expressie**-resultaat toe aan een variabele:

```
1 int temperatuursVerschil = temperatuurGisteren - temperatuurVandaag;
```

Hierbij zal de temperatuur uit de rechtse 2 variabelen worden uitgelezen, van elkaar worden afgetrokken en vervolgens bewaard worden in `temperatuursVerschil`.

Een ander voorbeeld van een **expressie**-resultaat toewijzen maar nu met literals:

```
1 int temperatuursVerschil = 21 - 25;
```

Uiteraard mag je ook combinaties van literals en variabelen gebruiken in je expressies:

```
1 int breedte = 15;  
2 int oppervlakte = 20 * breedte;
```

2.6.2 Operators en operanden

Om expressies te gebruiken hebben we ook zogenaamde **operators** nodig. Operators in C# zijn de wiskundige bewerkingen zoals optellen, aftrekken, vermenigvuldigen en delen. Deze volgen de klassieke wiskundige regels van **volgorde van berekeningen**:

1. **Haakjes**
2. **Vermenigvuldigen, delen en modulo**: $*$, $/$, $\%$ (rest na deling, ook modulo genoemd).
3. **Optellen en aftrekken**: $+$ en $-$



We spreken over operators en **operanden**. Een operand is het element dat we links en/of rechts van een operator zetten. In de som $3+2$ zijn 3 en 2 de operanden, en + de operator. In dit voorbeeld spreken we van een **binaire operator** omdat er twee operanden zijn.

Er bestaan ook **unaire operators** die maar 1 operand hebben. Denk bijvoorbeeld aan de - operator om het teken van een getal om te wisselen: -6.

In hoofdstuk 5 zullen we nog een derde type operator ontdekken: de **ternaire operator** die met 3 operanden werkt!

Net zoals in de wiskunde kan je in C# met behulp van de haakjes verplichten het deel tussen de haakjes eerst te berekenen, ongeacht de andere operators en hun volgorde van berekeningen:

```
1 3+5*2 // zal 13 (type int) als resultaat geven
2 (3+5)*2 // zal 16 (type int) geven
```

Je kan nu complexe berekeningen doen door literals, operators en variabelen samen te voegen. Bijvoorbeeld om te weten hoeveel je op Mars zou wegen:

```
1 double gewichtOpAarde = 80.3; //kg
2 double gAarde = 9.81;
3 double gMars = 3.711;
4 double gewichtOpMars = (gewichtOpAarde/gAarde) * gMars; //kg
5 Console.WriteLine("Je weegt op Mars " + gewichtOpMars + " kg");
```

2.6.2.1 Modulo operator %

De modulo operator die we in C# aanduiden met % verdient wat meer uitleg. Deze operator zal als resultaat de gehele rest teruggeven wanneer we het linkse getal door het rechtse getal delen:

```
1 int rest = 7%2;
2 int resultaat2 = 10%5;
```

Lijn 1 resulteert in de waarde 1 die in rest wordt bewaard: 7 delen door 2 geeft 3 met rest 1. Lijn 2 zal 0 geven, want 10 delen door 5 heeft geen rest.

De modulo-operator zal je geregeld gebruiken om bijvoorbeeld te weten of een getal een veelvoud van iets is. Als de rest dan 0 is weet je dat het getal een veelvoud is van het getal waar je het door deelde.

Bijvoorbeeld om te testen of getal even is gebruiken we %2:

```
1 int getal = 1234234;
2 int rest = getal%2;
3 Console.WriteLine("Indien het getal als rest 0 geeft is deze even.");
4 Console.WriteLine("De rest is: " + rest);
```

2.6.2.2 Verkorte operator notaties

Heel vaak wil je de inhoud van een variabele bewerken en dan terug bewaren in de variabele zelf. Bijvoorbeeld een variabele vermenigvuldigen met 10 en het resultaat ervan terug in de variabele plaatsen. Hiervoor zijn enkele verkorte notaties in C#. Stel dat we een variabele `int getal` hebben:

| Verkorte notatie | Lange notatie | Beschrijving |
|------------------------|-------------------------------|--|
| <code>getal++;</code> | <code>getal = getal+1;</code> | variabele met 1 verhogen |
| <code>getal--;</code> | <code>getal = getal-1;</code> | variabele met 1 verlagen |
| <code>getal+=3;</code> | <code>getal = getal+3;</code> | variabele verhogen met een getal |
| <code>getal-=6;</code> | <code>getal = getal-6;</code> | variabele verminderen met een getal |
| <code>getal*=7;</code> | <code>getal = getal*7;</code> | variabele vermenigvuldigen met een getal |
| <code>getal/=2;</code> | <code>getal = getal/2;</code> | variabele delen door een getal |



Je zal deze verkorte notatie vaak tegenkomen. Ze zijn identiek aan elkaar en zullen dus je code niet versnellen. Ze zal enkel compacter zijn om te lezen. Bij twijfel, gebruik gewoon de lange notatie.



De verkorte notaties hebben ook een variant waarbij de operator links en de operand rechts staat. Bijvoorbeeld `--getal`. Beide doen het zelfde, maar niet helemaal. Je merkt het verschil in volgende voorbeeld:

```
1 int getal = 1;
2 int som = getal++; //som wordt 1, getal wordt 2
3 int som2 = ++som; //som2 wordt 2, som wordt 2
```

Als je de operator achter de operand zet (`som++`) dan zal eerst de waarde van de operand worden teruggegeven, vervolgens wordt deze verhoogd. Bij de andere (`++som`) is dat omgekeerd: eerst wordt de operand aangepast, vervolgens wordt nieuwe waarde als resultaat van de expressie teruggegeven.



Gegroet! Zet je helm op en let alsjeblieft goed op. Als je de volgende sectie goed begrijpt dan heb je al een grote stap vooruit gezet in de wondere wereld van C#.

Ik zei je al dat variabelen het hart van programmeren zijn. Wel, expressies zijn het bloedvatensysteem dat ervoor zorgt dat al je variabelen ook effectief gecombineerd kunnen worden tot wondermooie nieuwe dingen.

Succes!



De voorman verschijnt wanneer er iets beschreven wordt waar véél fouten op gemaakt worden, zelfs bij ervaren programmeurs. Opletten geblazen dus.

2.7 Expressiedatatypes

Lees deze zin enkele keren luidop voor, voor je verder gaat: **De types die je in je expressies gebruikt bepalen ook het type van het resultaat.** Als je bijvoorbeeld twee **int** variabelen of literals optelt zal het resultaat terug een **int** geven (klink logisch, maar lees aandachtig verder):

```
1 int result = 3 + 4;
```

Je kan echter geen kommagetallen aan **int** toewijzen. Als je dus twee **double** variabelen deelt is het resultaat terug een **double** en zal deze lijn een fout geven daar je probeert een **double** aan een **int** toe te wijzen:

```
1 int otherResult = 3.1 / 45.2; //dit is fout!!!
```

Bovenstaande code geeft volgende fout: “Cannot implicitly convert double to int.”

Let hier op!

2.7.1 But wait... it gets worse!

Wat als je een `int` door een `int` deelt? Het resultaat is terug een `int`. Je bent echter alle informatie na de komma kwijt. Kijk maar:

```
1 int getal1 = 9;
2 int getal2 = 2;
3 int result = getal1/getal2;
4 Console.WriteLine(result);
```

Er zal 4 op het scherm verschijnen! (niet 4.5 daar dat geen `int` is).

2.7.2 Datatypes mengen in een expressie

Wat als je datatypes mengt? Als je een berekening doet met bijvoorbeeld een `int` en een `double` dan zal C# het 'grootste' datatype kiezen. In dit geval een `double`.

Volgende code zal dus werken:

```
1 double result = 3/5.6;
```

Volgende code niet:

```
1 int result = 3/5.6;
```

En zal weer dezelfde fout genereren: *"Cannot implicitly convert type 'double' to 'int'. An explicit conversion exists (are you missing a cast?)"*

Wil je dus het probleem oplossen om 9 te delen door 2 en toch 4.5 te krijgen (en niet 4) dan zal je minstens 1 van de 2 literals of variabelen naar een `double` moeten omzetten.

Het voorbeeld van hierboven herschrijven we daarom naar:

```
1 int getal1 = 9;
2 double getal2 = 2.0; //slim he
3 double result = getal1/getal2;
4 Console.WriteLine(result);
```

En nu krijgen we wel 4.5 aangezien we nu een `int` door een `double` delen en C# dus ook het resultaat dan als een `double` zal teruggeven.



Begrijp je nu waarom dit een belangrijk deel was? Je kan snel erg foute berekeningen en ongewenste afrondingen krijgen indien je niet bewust omgaat met je datatypes. Laten we eens kijken of je goed hebt opgelet, het kan namelijk subtiel en ambetant worden in grotere berekeningen.

Stel dat ik afsprek dat je van mij de helft van m'n salaris krijgt¹. Ik verdien 10 000 euro per maand (*I wish*).

Ik stel je voor om volgende expressie te gebruiken om te berekenen wat je van mij krijgt:

```
1 double helft = 10000.0 * (1 / 2);
```

Hoeveel krijg je van me?

0.0 euro, MUHAHAHAHA!!!

Begrijp je waarom? De volgorde van berekeningen zal eerst het gedeelte tussen de haakjes doen:

- 1 delen door 2 geeft 0, daar we een **int** door een **int** delen en dus terug een **int** als resultaat krijgen.
- Vervolgens zullen we deze 0 vermenigvuldigen met 10000.0 waarvan ik zo slim was om deze in **double** te zetten. Niet dus. We vermenigvuldigen weliswaar een **double** (10000.0) met een **int**, maar die **int** is reeds 0 en we krijgen dus 0.0 als resultaat.

Als ik dus effectief de helft van m'n salaris wil afstaan dan moet ik de expressie aanpassen naar bijvoorbeeld:

```
1 double helft = 10000.0 * (1.0 / 2);
```

Nu krijgt het gedeelte tussen de haakjes een **double** als resultaat, namelijk 0.5 dat we dan kunnen vermenigvuldigen met het salaris om 5000.0 te krijgen, wat jij vermoedelijk een fijner resultaat vindt.

¹Voorgaande voorbeeld is gebaseerd op een oefening uit het handboek "Programmeren in C#" van Douglas Bell en Mike Parr, een boek dat werd vertaald door collega lector Kris Hermans bij de Hogeschool PXL. Als je de console-applicaties beu bent en liever leert programmeren door direct grafische Windows-applicatie te maken, dan raad ik je dit boek ten stelligste aan!

2.7.3 Constanten

Je zal het **const** keyword hier en daar in codevoorbeelden zien staan. Je gebruikt dit om aan te geven dat een variabele onveranderlijk is én niet per ongeluk kan aangepast worden. Door dit keyword voor de variabele declaratie te plaatsen zeggen we dat deze variabele na initialisatie niet meer aangepast kan worden.

Volgende voorbeeld toont in de eerste lijn hoe je het **const** gebruikt. De volgende lijn zal dankzij dit keyword een error geven reeds bij het compileren en jou dus waarschuwen dat er iets niet klopt.

```
1 const double G_AARDE = 9.81;  
2 G_AARDE = 10.48; //ZAL ERROR GEVEN
```

Merk op hoe we de **const** variabelen een identifier geven: deze zetten we in ALLCAPS. Hierbij gebruiken we een liggend streepjes om het onderscheid tussen de onderlinge woorden aan te geven. Dit is geen verplichting, maar gewoon een aanbeveling.



Constanten in code worden ook soms **magic numbers** genoemd. De reden hiervoor is dat ze vaak plotsklaps ergens in de code voorkomen, maar wel op een heel andere plek werden gedeclareerd. Hierdoor is het voor de ontwikkelaar niet altijd duidelijk wat de variabele juist doet. Het is daarom belangrijk dat je goed nadenkt over het gebruik van magic numbers én deze zeer duidelijke namen geeft.

Er worden vele *filosofische oorlogen* gevoerd tussen ontwikkelaars over de plek van magic numbers in code. In de C/C++ tijden werden deze steeds bovenaan aan de start van de code gegroepeerd. Op die manier zag de ontwikkelaar in één oogopslag alle belangrijke variabelen en konden deze ook snel aangepast worden. In C# prefereert men echter om variabelen zo dicht mogelijk bij de plek waar ze nodig zijn te schrijven, dit verhoogt de *modulariteit* van de code: je kan sneller een flard code kopiëren en op een andere plek herbruiken.

De applicaties die wij in dit boek ontwikkelen zijn niet groot genoeg om over te debatteren. Veel bedrijven hanteren hun eigen coding guidelines en het gebruik, naamgeving en plaatsing van magic numbers zal zeker daarin zijn opgenomen.

2.8 Solutions en projecten

Het wordt tijd om eens te kijken hoe Visual Studio jouw code juist organiseert wanneer je een nieuw project start. Zoals je al hebt gemerkt in de solution Explorer wordt er meer aangemaakt dan enkel een Program.cs codebestand. Visual Studio werkt volgens volgende hiërarchie:

1. Een **solution** is een folder waarbinnen **één of meerdere projecten** bestaan.
2. Een **project** is een verzameling (code)bestanden die samen een specifieke functionaliteit vormen en kunnen worden gecompileerd tot een uitvoerbaar bestand, bibliotheek, of andere vorm van output (we vereenvoudigen bewust het concept project in dit handboek).

Wanneer je dus aan de start van een nieuwe opdracht staat en in VS kiest voor “Create a new project” dan zal je eigenlijk aan een nieuwe solution beginnen met daarin één project.

Je bent echter niet beperkt om binnen een solution maar één project te bewaren. Integendeel, vaak kan het handig zijn om meerdere projecten samen te houden. Ieder project bestaat op zichzelf, maar wordt wel logisch bij elkaar gehouden in de solution. Dat is ook de reden waarom we vanaf de start hebben aangeraden om nooit het vinkje *“Place solution and project in the same directory”* aan te duiden.

2.8.1 Folderstructuur van een solution

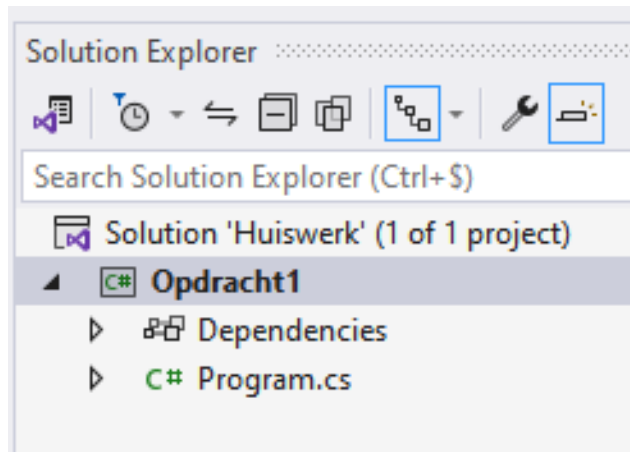
Wanneer je in VS een nieuw project start ben je niet verplicht om de “Project name” en “Solution name” dezelfde waarde te geven. Je zal wel merken dat bij het invoeren van de “Project name” de “Solution name” dezelfde invoer krijgt. Je mag echter vervolgens perfect de “Solution name” aanpassen.

Stel dat we een nieuw VS project aanmaken met volgende informatie:

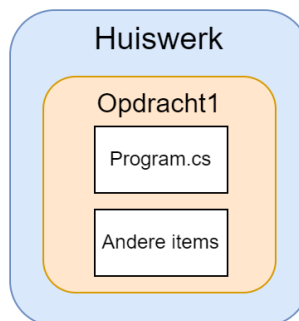
1. Naam van het project = **Opdracht1**
2. Naam van de solution = **Huiswerk**

En plaatsen deze in de folder C : \ Temp.

Wanneer we het project hebben aangemaakt en de Solution Explorer bekijken zien we volgende beeld :

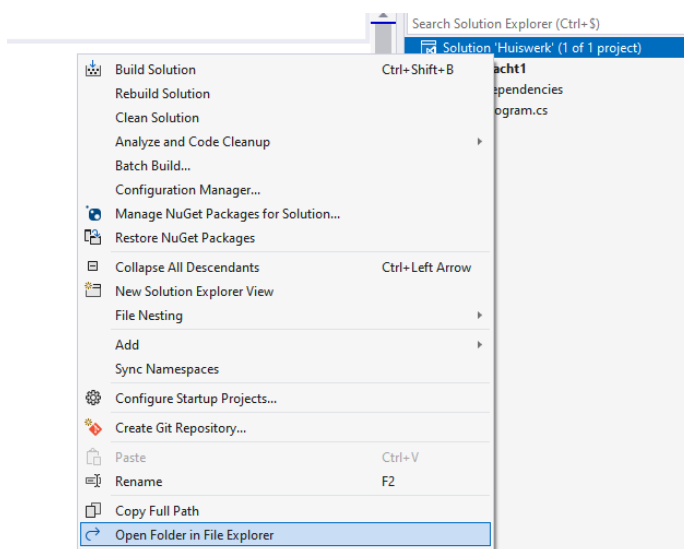


Figuur 2.4: Je ziet duidelijk een hiërarchie: bovenaan de solution *Huiswerk*, met daarin een project *Opdracht1*, gevuld met informatie zoals het *Program.cs* bestand. Deze hiërarchie zal je ook terugzien als je via de verkenner vervolgens naar de aangemaakte folder zou gaan.



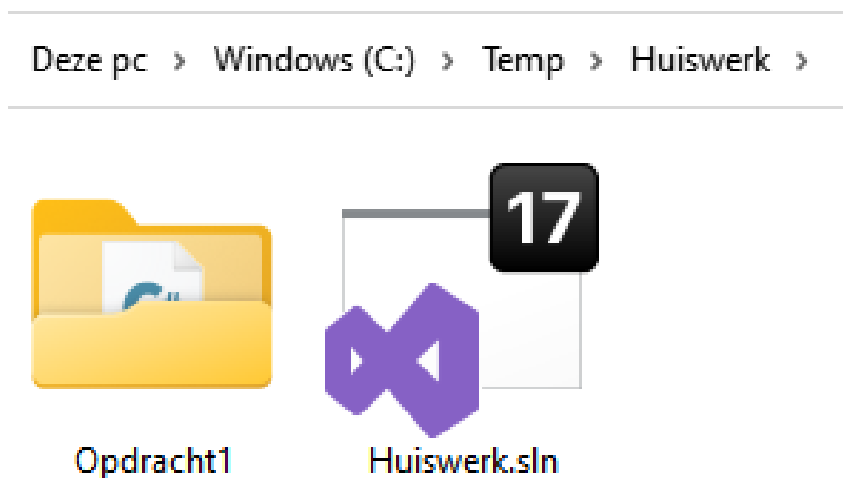
Figuur 2.5: De hiërarchie anders voorgesteld.

Rechterklik nu op de solution en kies “Open folder in file explorer”. Je kan deze optie kiezen bij eender welk item in de solution explorer. Het zal er voor zorgen dat de verkenner wordt geopend op de plek waar het item staat waar je op rechterklikte. Op die manier kan je altijd ontdekken waar een bestand of of folder zich fysiek bevindt op je harde schijf.



Figuur 2.6: Tip: rechtermklikken in veel programma's geeft je vaak toegang tot meer geavanceerde commando's, zo ook in VS.

We zien nu een tweede belangrijke aspect dat we in deze sectie willen uitleggen: **Een solution wordt in een folder geplaatst met dezelfde naam én bevat één .sln bestand. Binnenin deze folder wordt een folder aangemaakt met de naam van het project.** Je folderstructuur volgt dus flink de structuur van je solution in VS.



Figuur 2.7: Merk op dat je mogelijk ook nog verborgen bestanden zal zien, afhankelijk van de instellingen van je verkenner.

Je kan dus je volledige solution, inclusief het project, openen door in deze folder het .sln bestand te selecteren. **Dit .sln bestand zelf bevat echter geen code.**



Die laatste zin heeft als gevolg dat je de **hele folderstructuur** moet verplaatsen indien je aan je solution op een andere plek wilt werken. Open gerust eens een .sln-bestand in notepad en je zal zien dat het bestand onder andere oplijst waar het onderliggende project zich bevindt. Wil je dus je solution doorgeven of mailen naar iemand, zorg er dan voor je de hele folderstructuur doorgeeft, inclusief het .sln bestand en alles folders die er bij horen.

2.8.2 Folderstructuur van een project

Laten we nu eens kijken hoe de folderstructuur van het project zelf is. Rechterklik deze keer op het project in de solution explorer (**Opdracht1**) en kies weer “Open folder in file explorer”.

Hier staat een herkenbaar bestand! Inderdaad, het *Program.cs* codebestand. In dit bestand staat de actuele code van Opdracht1.

Voorts zien we ook een .csproj bestand genaamd *Opdracht1*. Net zoals het .sln bestand zal dit bestand beschrijven welke bestanden én folder(s) deel uitmaken van het huidige project. Je kan dit bestand dus ook openen vanuit de verkenner en je zal dan je volledige project zien worden ingeladen in Visual Studio.

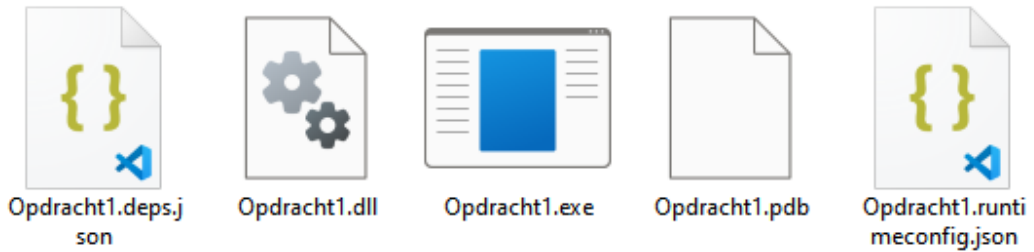


Een .cs-bestand rechtstreeks vanuit de verkenner openen werkt niet zoals je zou verwachten. VS zal weliswaar de inhoud van het bestand tonen, maar je kan verder niets doen. Je kan niet compileren, debuggen, enz. De reden is eenvoudig: een .cs bestand op zichzelf is nutteloos. Het heeft pas een bestaansreden wanneer het wordt geopend in een project. Het project zal namelijk beschrijven hoe dit specifieke bestand juist moet gebruikt worden in het huidige project.

2.8.2.1 De bin-folder

De “obj” folder ga ik in dit handboek negeren. Maar kijk eens wat er in de “bin” folder staat?! Een folder genaamd “**debug**”. In deze folder zal je de gecompileerde (debug-)versie van je huidige project terecht komen. Je zal wat moeten doorklikken tot de *binnenste folder* (die de naam van de huidige .net versie bevat waarin je compileert).

Deze pc > Windows (C:) > Temp > Huiswerk > Opdracht1 > bin > Debug > net8.0

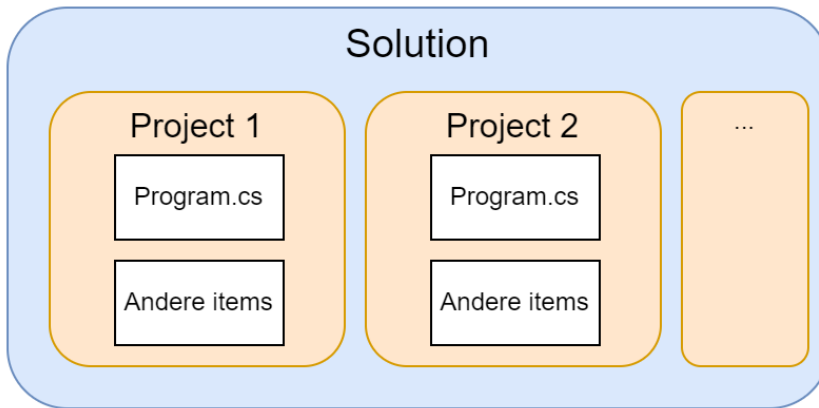


Figuur 2.8: Inhoud van bin/debug/net8.0 nadat project werd gecompileerd

Je kan in principe vanuit deze folder ook je gecompileerde project uitvoeren door te dubbelklikken op *Opdracht1.exe*. Je zal echter merken dat het programma ogenblikkelijk terug afsluit omdat het programma aan het einde van de code altijd afsluit. Voeg daarom volgende lijn code toe onderaan in je `Main: Console.ReadLine()`. Het programma zal nu pas afsluiten wanneer je op Enter hebt gedrukt en de gecompileerde versie kan dus nu vanuit de verkennen gestart worden, hoera!

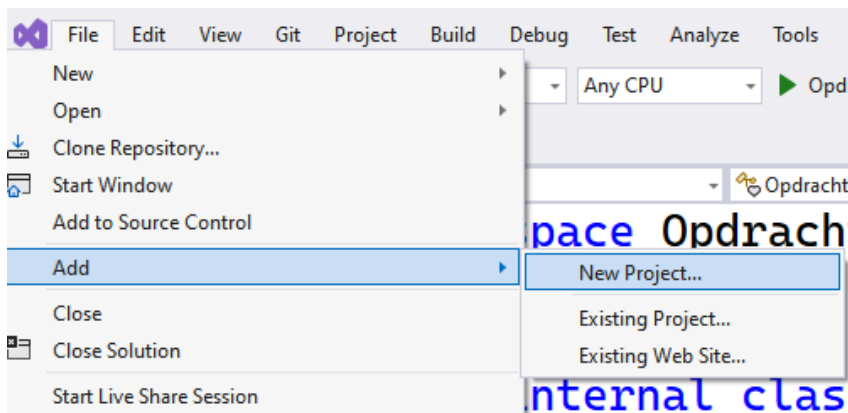
Merk op dat je de volledige inhoud van deze folder moet meegeven indien je je gecompileerde resultaat aan iemand wilt geven om uit te voeren.

2.8.3 Meerdere projecten



Figuur 2.9: Er is geen limiet op het aantal projecten in 1 solution. De enige beperking is de kracht van je computer.

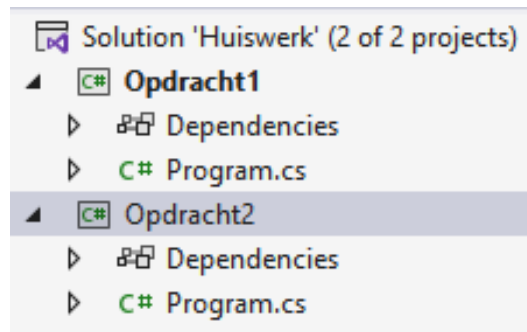
Ik zei net dat een solution meerdere projecten kan bevatten. Maar hoe voeg je een extra project toe? Terwijl je huidige solution open is (waar je een project wenst aan toe te voegen) kies je in het menu voor *File->Add->New project...*



Figuur 2.10: Ook “Existing project...” is een handige actie om te kennen!

Je moet nu weer het klassieke proces doorlopen om een console-project aan te maken. Alleen ontbreekt deze keer het “Solution name” tekstveld, daar dit reeds gekend is.

Wanneer je klaar bent zal je zien dat in de solution Explorer een tweede project is verschenen. Als we de folderstructuur van onze solution opnieuw bekijken, zien we dat er een nieuwe folder (Opdracht2) is verschenen met een eigen Program.cs en .csproj-bestand.



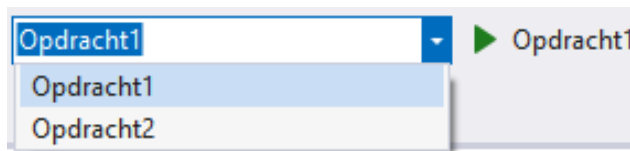
Figuur 2.11: Gelukkig kan je zaken dichtklappen m.b.v. driehoekjes naast iedere item.

Nu rest ons nog één belangrijke stap: **selecteren welk project moet gecompileerd en uitgevoerd worden**. In de solution explorer kan je zien welk het *actieve* project is, namelijk het project dat vet gedrukt staat.

Je kan nu op 2 manieren kiezen welk project moet uitgevoerd worden:

Manier 1: Rechterklik in de Solution Explorer op het actief te zetten project en kies voor “Set as startup project.”

Manier 2: Bovenaan, links van de groene “compiler/run” knop, staat een selectieveld met het actieve project. Je kan hier een andere project selecteren.



Figuur 2.12: Tijd om naar Opdracht2 over te schakelen.



Controleer altijd goed dat je in het juiste Program.cs bestand bent aan het werken. Je zou niet de eerste zijn die maar niet begrijpt waarom de code die je invoert z’n weg niet vindt naar je debugvenster. Inderdaad, vermoedelijk heb je het verkeerde Program.cs bestand open *of* heb je het verkeerde actieve project gekozen.



Ook nu reeds heb je mogelijk interesse in meerdere projecten in 1 solution. Je kan nu perfect je opdrachten groeperen onder 1 solution, maar toch iedere opdracht mooi gescheiden houden. In de echte wereld gebruikt men meerdere projecten in 1 solution om het overzicht te bewaren en alles zo modulair mogelijk aan te pakken. Denk maar aan een solution met een projecten dat de (unit)testen bevat, een project voor de *frontend*, en nog een project voor de *backend*.

2.8.3.1 Delen met de oma

Om een gecompileerde .NET applicatie te kunnen uitvoeren op een computer heb je nog een **.NET runtime** nodig. Gebruikers die geen Visual Studio hebben geïnstalleerd hebben deze runtime meestal niet op hun systeem.

Wil je dus dat je oma kan genieten van jouw laatste creatie, zorg er dan voor dat ze de juiste .NET runtime heeft draaien. Je zal haar hier wat mee moeten helpen want je moet de runtime installeren² voor die versie *waar tegen jouw applicatie is gecompileerd*.

² Je kan alle .NET runtimes hier terugvinden: dotnet.microsoft.com/en-us/download/dotnet